New methods for controlling timing channels

Andrew Myers
Cornell University
(with Danfeng Zhang, Aslan Askarov)
Timing channels

The adversary can learn (a lot) from timing measurements.

Known to exist
Hard to detect
Hard to prevent except in special cases
undetectable threat of unknown importance + lack of feasible defenses
A few timing attacks

- Network timing attacks
  - RSA keys leaked by decryption time, measured across network [Brumley&Boneh’05]
  - Load time of web page reveals login status, size and contents of shopping cart [Bortz&Boneh’07]

- Cache timing attacks
  - AES keys leaked by timing memory accesses [Osvik et al’06] from ~300 (!) encryptions

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

- Timing channels: a serious threat
The problem

- Timing may encode any secrets it depends on
- Strong adversary: able to affect system timing (coresident code, by adding load,...)
Timing channel mitigation

- Some standard ideas:
  - Add random delays ⇒ lower bandwidth, linear leakage
  - Delay to worst-case time ⇒ poor performance
  - Input blinding ⇒ applicable only to cryptography

- New idea: **predictive mitigation**
  - Applies to general computation
  - Leakage asymptotically sublinear over time
  - Effective in practice
  - Applicable at system and language level
Variations $\rightarrow$ leakage

Leakage in bits $= \log_2 N$

A bound on:
- mutual information (Shannon entropy)
- min-entropy

$N$ possible observations by the adversary
Black-box predictive mitigation

Issues events according to schedules
Prediction by doubling

**predictions:** when mitigator expects to deliver events

events

Mitigator starts with a fixed schedule $S$

$S(n)$ – prediction for $n$th event
Example: Doubling

When event comes before or at the prediction – delay the event

little information leaked
Example: Doubling

Adversary observes mispredictions $\Rightarrow$ information leaked
New fixed schedule $S_2$ penalizes the event source
Example: Doubling

**Epoch**: period of time during which mitigator meets all predictions

Within epoch, output times can be predicted by adversary too!
Quantifying leakage

• Variations within one epoch = $M+1 = O(T)$
• Over $N$ epochs? $(M+1)^N$

Leakage $\leq N \log(M+1)$ bits $= O(N \log T)$ bits

• Leakage with doubling scheme:

$N = O(\log T)$
leakage $\leq O(\log^2 T)$
Adaptive transitions

• If predictions become too conservative, events are delayed
  • queueing ⇒ no mispredictions

• Idea. if under misprediction “budget”, force an epoch change:
  • dump queued events
  • generate a new schedule with better performance
Using public information

- Simple black-box model [CCS’10]
  - Fixed schedule in each epoch – too conservative for interactive systems

- Generalized prediction [CCS’11]
  - Fixed prediction algorithm implementing a deterministic function of public information
    - Schedule is calculated *dynamically* within epoch
    - Algorithm changed at mispredictions
Exploitable public information

Using public information improves predictions for networked applications

• Public payloads in requests, such as URLs
  
  www.example.com/index.html vs. www.example.com/background.gif

• Time of input request
Evaluation

Real-world web applications (with HTTP(S) proxy)
Mitigating Web proxy

Demo
Experiments with Web applications

Mitigating department homepage via HTTP

(49 different requests)

• Different prediction schemes trade off security vs. performance. With HOST+URLTYPE scheme:
  • \(~30\%\) latency overhead
  • < 850bits for 100,000 inputs
Experiments with Web applications

Mitigating department webmail server via HTTPS

- At most 300 bits for 100,000 inputs
- At most 450 bits for 32M inputs
  (1 input/sec for one year)

Performance

Security
Related work

• Timing mitigation for cryptographic operations
  [Kocher 96, Kopf & Durmuth 09, Kopf & Smith 10]
  • Assumes input blinding

• NRL Pump/Network Pump [Kang et. al. 93, 96]
  • Addresses covert channels from input acks
  • Linear bound

• Information theory community [Hu 91, Giles&Hajek 02]
  • Timing mitigation based on random delays
  • Linear bound
Why **language-level mitigation**?

- What about the coresident adversary who can time accesses to memory?
  - AES keys leaked by timing memory accesses from ~300 (!) encryptions [Osvik et al 06]
  - A real problem for cloud computing...

- How can programmer know whether program has timing channels?
  - Idea: provide a static analysis (e.g., type system) that verifies bounded leakage.
    - and incorporate predictive mitigation!
Security policies

• Security policy lattice
  - Information has label describing intended confidentiality
  - In general, the labels form a lattice
  - For this talk, a simple lattice:
    - L=public, H=secret
    - H should not flow to L

• Adversary powers
  - Sees contents of low (L) memory (storage channel)
  - Sees timing of updates to low memory (timing channel)
A timing channel

if (h)
    sleep(1);
else
    sleep(2);
A subtle example

```c
if (h1)
    h2=l1;
else
    h2=l2;
l3=l1;
```

Data cache affects timing!
Beneath the surface

if (h1)
  h2=l1;
else
  h2=l2;
l3=l1;

interface?
guarantees?

compiler optimizations

data/instruction cache

branch target buffer
data/instruction TLB
A language-level abstraction

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

\[(x := e) \begin{bmatrix} l_r, l_w \end{bmatrix}\]

Machine environment: state affecting timing but invisible at language level

- Does not include language-visible state (memory)

- Machine environment is logically partitioned by security level (e.g. high cache vs. low cache)
Read label

\[(x := e)_{[l_r, l_w]}\]

abstracts how machine environment affects time taken by next language-level step.

= upper bound on influence
Write label

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

abstracts how machine environment is affected by next language-level step

= lower bound on effects
Security properties

- Language implementation must satisfy three (formally defined) properties:
  1. Read label property
  2. Write label property
  3. Single-step noninterference: no leaks from high environment to low environment

- Realizable on commodity HW (no-fill mode)

- Provides guidance to designers of future secure architectures
Type system

• We analyze programs using an information flow type system that tracks timing

\[ c : T \Rightarrow \text{time to run } c \text{ depends on information at (at most) label } T \]

• Read and write labels are key
  • can be generated by analysis, inference, programmer...

Examples:

\[
\begin{align*}
  c[H,\ell_{w}] : H \\
  \text{sleep}(h) : H \\
  (x := y)[L,L] : L \\
  \text{if (} h_1 \text{)} \\
  \quad (h_2 := l_1)[L,H] ; \\
  \quad \text{else} \\
  \quad (h_2 := l_2)[L,H] ; \\
  \quad (l_2 := l_1)[L,L] \\
\end{align*}
\]

low cache read cannot be affected by \( h_1 \)
Formal results

- Memory and machine environment noninterference:
  A well-typed program *without* use of mitigation leaks nothing via timing channels

![Diagram](image)

- Before execution
- After execution
Language-level mitigation

- Executes $s$ but adds time using predictive mitigation
  - New expressive power:
    ```
    \text{sleep}(h):H \quad \text{but} \quad \text{mitigate}(l) \begin{cases} \text{sleep (h)} \end{cases}:L
    ```
  - Result: well-typed program using mitigate has bounded leakage (e.g., $O(\log^2 T)$)
Evaluation Setup

- Simulated architecture satisfying security properties with statically partitioned cache and TLB

<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>

- Implemented on SimpleScalar simulator, v.3.0e
Web login example

- Valid usernames can be learned via timing [Bortz&Boneh 07]
- Secret
  - MD5 digest of valid (username, password) pairs
- Inputs
  - 100 different (username, password) pairs
Login behavior

valid usernames

100  50  10  no mitigation

login time (in # of clock cycles)

39600  39800  40000  40200  40400

with mitigation

87000  87015  87030  87045
Performance

- nopar: unmodified hardware
- moff: secure hardware, no mitigation
- mon: secure 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>
RSA

- RSA reference implementation
- Secret: private keys
- Inputs: different encrypted messages
RSA behavior

![Graph showing RSA behavior with and without mitigation]
Conclusions

- We should care about timing channels.

- Sources of optimism:
  - **Predictive mitigation**, a new dynamic mechanism for controlling leakage
  - **Read and write labels** as a clean, general abstraction of hardware timing behavior, enabling software/hardware codesign and...
  - Static analysis of timing behavior with strong guarantees of bounded information leakage.