Suppressing the Oblivious RAM Timing Channel

While Making Information Leakage and Program Efficiency Trade-offs

Chris Fletcher\textsuperscript{1}, Ling Ren\textsuperscript{1}, Xiangyao Yu\textsuperscript{1}, Marten V. Dijk\textsuperscript{2}, Omer Khan\textsuperscript{2}, Srinivas Devadas\textsuperscript{1}

\textsuperscript{1}Massachusetts Institute of Technology \quad \textsuperscript{2}University of Connecticut
Timing attacks in secure processors

Secure cloud processor

Enc(Result)

Enc(Data)

Program

Data

Enc(Data)

Enc(Result)

DRAM
Timing attacks in secure processors

Data dependent program behavior → timing channels
Timing attacks in secure processors

Data dependent program behavior $\rightarrow$ timing channels

Secure cloud processor

- Enc(Read)
- Enc(Data)
- Enc(Result)

e.g., SVF, ISCA’12
Timing attacks in secure processors

Data dependent program behavior $\rightarrow$ timing channels

Secure cloud processor

e.g., SurfNoc, ISCA’13

Program

Data

Enc(Data)

Enc(Result)

e.g., SVF, ISCA’12

Enc(Enc(Data))

Enc(Enc(Result))
Timing attacks in secure processors

Data dependent program behavior $\rightarrow$ timing channels
Timing attacks in secure processors

Data dependent program behavior $\rightarrow$ timing channels

Secure cloud processor

2 papers in HPCA’14!

Enc(Result)

Enc(Data)

e.g., SVF, ISCA’12

e.g., RP/PL $, ISCA’07

e.g., SurfNoc, ISCA’13

2 papers in HPCA’14!
Timing attacks in secure processors

Data dependent program behavior $\rightarrow$ timing channels

Timing channels $\rightarrow$ information leakage
Leakage ↔ efficiency spectrum

- Low leakage
  - Low efficiency

- High leakage
  - High efficiency
Leakage ← → efficiency spectrum

Low leakage
Low efficiency

High leakage
High efficiency

Non-interference
[e.g., SurfNoc, ISCA13]
(simplest case: static partitioning)
Leakage ↔ efficiency spectrum

Low leakage
Low efficiency

High leakage
High efficiency

Non-interference
[e.g., SurfNoc, ISCA13]
(simplest case: static partitioning)

Most proposals
[e.g., No-Mo caches, TACO12]
Leakage $\leftrightarrow$ efficiency spectrum

Our proposal: Leakage aware processors
- Hardware mechanisms + information theory to quantify leakage [Smith, FOSSACS09]
- Architect chip so that as leakage increases, efficiency increases
Leakage ↔ efficiency spectrum

Low leakage
Low efficiency

High leakage
High efficiency

Non-interference
[e.g., SurfNoc, ISCA13]
(simplest case: static partitioning)

Most proposals
[e.g., No-Mo caches, TACO12]

Our proposal: Leakage aware processors

- Hardware mechanisms + information theory to quantify leakage [Smith, FOSSACS09]
- Architect chip so that as leakage increases, efficiency increases
DRAM protection

Processor

Program, Data

$->

Data

Address

Op (R/W)

Access times

DRAM
• **Oblivious RAM (ORAM)**
  – Make main memory access patterns look random
  – Recently built into hardware [Ren et al., ISCA’13; Phantom, CCS’13]
DRAM protection

- **Oblivious RAM (ORAM)**
  - Make main memory access patterns look random
  - Recently built into hardware [Ren et al., ISCA’13; Phantom, CCS’13]

- ORAM accesses more expensive than DRAM accesses
- Dummy ORAM access == Real ORAM access
No leakage: pick a static rate offline

- E.g., “access ORAM every 100 cycles”
No leakage: pick a static rate offline

- E.g., “access ORAM every 100 cycles”
- Access made when not needed $\rightarrow$ dummy access \textcolor{red}{(power++)}
- Access needed before made $\rightarrow$ program waits \textcolor{red}{(performance--)}
No leakage: pick a static rate offline

- E.g., “access ORAM every 100 cycles”
- Access made when not needed $\Rightarrow$ dummy access (power++)
- Access needed before made $\Rightarrow$ program waits (performance--)

- Problem: Rate can change across inputs
No leakage: pick a static rate offline

- E.g., “access ORAM every 100 cycles”
- Access made when not needed $\Rightarrow$ dummy access (\textit{power++})
- Access needed before made $\Rightarrow$ program waits (\textit{performance--})

- Problem: Rate can change across inputs

![Graph showing rate over time for perl](image)

- Problem: A single input may need multiple rates

![Graph showing rate over time for astar](image)
We need data-dependent rates!

But … data-dependent rates leak information.

How do we leak enough without leaking everything?
Bounding leakage

- **Key Insight** [Kopf, CSF10; Smith, FOSSACS09]
  
  $N$ timing traces $\rightarrow \leq \log_2(N)$ secret bits leaked
Bounding leakage

• **Key Insight** [Kopf, CSF10; Smith, FOSSACS09]
  \[ N \text{ timing traces} \rightarrow \leq \log_2(N) \text{ secret bits leaked} \]

• **ORAM timing traces**
  – Timing trace = unique ORAM rate signal

- Access (y/n)
- Time →
Bounding leakage

• **Key Insight** [Kopf, CSF10; Smith, FOSSACS09]

  \[N \text{ timing traces} \rightarrow \leq \log_2(N) \text{ secret bits leaked}\]

• **ORAM timing traces**
  – Timing trace = unique ORAM rate signal

• **ORAM leakage** – static, periodic rates

![Diagram](image)
Bounding leakage

- **Key Insight** [Kopf, CSF10; Smith, FOSSACS09]
  \[ N \text{ timing traces } \rightarrow \leq \log_2(N) \text{ secret bits leaked} \]

- **ORAM timing traces**
  - Timing trace = unique ORAM rate signal

- **ORAM leakage – static, periodic rates**
Bounding leakage

- **Key Insight** [Kopf, CSF10; Smith, FOSSACS09]
  
  \[ N \text{ timing traces } \rightarrow \leq \log_2(N) \text{ secret bits leaked} \]

- **ORAM timing traces**
  - Timing trace = unique ORAM rate signal

- **ORAM leakage – static, periodic rates**

Same signal always $\rightarrow N = 1$ trace $\rightarrow \log_2(1) = 0$ bits leaked

Non-inference leaks 0 bits!
Bounding leakage (2)

- ORAM leakage – no protection
Bounding leakage (2)

- ORAM leakage – no protection

```
EvilProgram():
for (i = 0, 1, ... |D|) {
    if (D[i]) miss LLC
    else       pause
}
```
Bounding leakage (2)

- ORAM leakage – no protection

```
EvilProgram():
    for (i = 0, 1, ... |D|) {
        if (D[i]) miss LLC
        else pause
    }
```

Time = 0 1 2 3

D[3:0] = 0000 → ________
Bounding leakage (2)

- ORAM leakage – no protection

EvilProgram():
for (i = 0, 1, … |D|) {
    if (D[i]) miss LLC
    else pause
}

Time = 0 1 2 3

D[3:0] = 0000 →
D[3:0] = 1000 →
Bounding leakage (2)

- ORAM leakage – no protection

```plaintext
EvilProgram()

for (i = 0, 1, ..., |D|) {
    if (D[i]) miss LLC
    else      pause
}
```

Time = 0 1 2 3

D[3:0] = 0000 \rightarrow

D[3:0] = 1000 \rightarrow

D[3:0] = 0100 \rightarrow
Bounding leakage (2)

- ORAM leakage – no protection

EvilProgram():
for (i = 0, 1, ... |D|) {
  if (D[i]) miss LLC
  else         pause
}

Time =

<table>
<thead>
<tr>
<th>0 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
<tr>
<td>_______</td>
</tr>
</tbody>
</table>

D[3:0] = 0000 →
D[3:0] = 1000 →
D[3:0] = 0100 →
D[3:0] = 1101 →
Bounding leakage (2)

- ORAM leakage – no protection

EvilProgram:
for (i = 0, 1, … |D|) {
  if (D[i]) miss LLC
  else      pause
}

Time = 0 1 2 3

D[3:0] = 0000 → \\
D[3:0] = 1000 → \\
D[3:0] = 0100 → \\
D[3:0] = 1101 → \\

T time → N = 2^T traces → log_2(2^T) = T bits leaked

Leakage grows linearly with run time!
Our approach: limit # traces $\rightarrow$ limit leakage

$2^T$ traces $\rightarrow$ poor security

1 trace $\rightarrow$ poor efficiency

$[2^{16}$ to $2^{32}$ traces $\rightarrow > 30\%$ efficiency over static$]$
Architectural overview

Secure processor

Pipeline, caches

ORAM Controller

Rate Learner

Leakage Knobs (E, R)
Architectural overview

1.) Discretize time into epochs (similar to [Askarov, CCS’10])
1.) Discretize time into epochs (similar to [Askarov, CCS’10])
2.) At the end of each epoch, choose a new static rate
**Architectural overview**

1.) Discretize time into epochs (similar to [Askarov, CCS’10])
2.) At the end of each epoch, choose a new static rate

- **Parameters**
  - **E:** Epoch growth rate \((\text{Epoch}_i = E \times \text{Epoch}_{i-1})\)
  - **R:** # of ORAM rates (each chosen offline) to consider for each epoch
• Leakage
  - $E$, 1\textsuperscript{st} epoch’s length, max runtime $\rightarrow$ \# epochs
  * max runtime = “big enough for all programs”
Architectural overview (2)

- **Leakage**
  - \( E \), 1\(^{st} \) epoch’s length, max runtime \( \rightarrow \) \# epochs
    * max runtime = “big enough for all programs”

- \( R(\# \text{epochs}) \) traces
- Leakage \( \leq \log_2(R(\# \text{epochs})) = (\# \text{epochs}) \times \log_2(R) \)
• Leakage
  – **E**, 1\textsuperscript{st} epoch’s length, max runtime $\rightarrow$ \# epochs
    * max runtime = “big enough for all programs”

• $R(\# \text{epochs})$ traces

• Leakage $\leq \log_2(R(\# \text{epochs})) = (\# \text{epochs}) \times \log_2(R)$

+ Leakage grows logarithmically w/ time  \hspace{1cm} - Doesn’t adapt quickly
Secure processor

Pipeline, caches

ORAM Controller

Rate Learner

E R

Time →

Architectural overview (3)
Architectural overview (3)

- E, R determine leakage
**Architectural overview (3)**

- **E**, **R** determine leakage

- **E**, **R** impact efficiency orthogonally
  - Decrease **E** for programs with more phases
  - Increase **R** for programs with more memory behaviors
Architectural overview (3)

- $E, R$ determine leakage

- $E, R$ impact efficiency orthogonally
  - Decrease $E$ for programs with more phases
  - Increase $R$ for programs with more memory behaviors
• **E, R** determine leakage

• **E, R** impact efficiency orthogonally
  • Decrease **E** for programs with more phases
  • Increase **R** for programs with more memory behaviors
Architectural overview (3)

- **E, R** determine leakage

- **E, R** impact efficiency orthogonally
  - Decrease **E** for programs with more phases
  - Increase **R** for programs with more memory behaviors
Architectural overview (3)

- \( E, R \) determine leakage

- \( E, R \) impact efficiency orthogonally
  - Decrease \( E \) for programs with more phases
  - Increase \( R \) for programs with more memory behaviors
Architectural overview (3)

- $E, R$ determine leakage

- $E, R$ impact efficiency orthogonally
  - Decrease $E$ for programs with more phases
  - Increase $R$ for programs with more memory behaviors

- User specifies leakage limit $\rightarrow$ server chooses $E, R$
• **E, R** determine leakage

• **E, R** impact efficiency orthogonally
  • Decrease $E$ for programs with more phases
  • Increase $R$ for programs with more memory behaviors

• User specifies leakage limit $\rightarrow$ server chooses $E, R$
Architectural overview (3)

- **E, R** determine leakage

- **E, R** impact efficiency orthogonally
  - Decrease **E** for programs with more phases
  - Increase **R** for programs with more memory behaviors

- User specifies leakage limit → server chooses **E, R**
Architectural overview (4)

- **How to choose rates: HW rate learner**
  - Passively monitors execution
  - Predicts “offered load” to ORAM
  - Maps prediction to 1 of $R$ rates (each chosen offline)
• How to choose rates: HW rate learner
  – Passively monitors execution
  – Predicts “offered load” to ORAM
  – Maps prediction to 1 of \( R \) rates (each chosen offline)
• **How to choose rates: HW rate learner**
  – Passively monitors execution
  – Predicts “offered load” to ORAM
  – Maps prediction to 1 of \( R \) rates (each chosen offline)
• **How to choose rates: HW rate learner**
  - Passively monitors execution
  - Predicts “offered load” to ORAM
  - Maps prediction to 1 of $R$ rates (each chosen offline)

• Small circuit (< 50B storage)
• Self contained: listens to LLC – ORAM controller queue
Evaluation

• Single core/SPEC app, 1 MB LLC
• 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
• Measure overall performance and on-chip power
Evaluation

- Single core/SPEC app, 1 MB LLC
- 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
- Measure overall performance and on-chip power

- Each experiment runs for 200-250 billion instructions
Evaluation

- Single core/SPEC app, 1 MB LLC
- 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
- Measure overall performance and on-chip power

- Each experiment runs for 200-250 billion instructions

- We compare against:
  - Insecure systems (no ORAM)
  - ORAM w/o timing protection
  - ORAM w/ various static rates
Evaluation

• Single core/SPEC app, 1 MB LLC
• 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
• Measure overall performance and on-chip power

• Each experiment runs for 200-250 billion instructions

• We compare against:
  – Insecure systems (no ORAM)
  – ORAM w/o timing protection
  – ORAM w/ various static rates

• Sweep knobs E, R
Evaluation

- Single core/SPEC app, 1 MB LLC
- 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
- Measure overall performance and on-chip power

- Each experiment runs for 200-250 billion instructions

- We compare against:
  - Insecure systems (no ORAM)
  - ORAM w/o timing protection
  - ORAM w/ various static rates

- Sweep knobs $E$, $R$

- First epoch = $2^{30}$ cycles
- Max runtime = $2^{62}$ cycles (>100 years)

\[
\text{Leakage} = \# \text{epochs} \times \log_2(R)
\]

\[
\text{\# epochs} = \frac{62 - 30}{\log_2(E)}
\]
Evaluation

• Single core/SPEC app, 1 MB LLC
• 1484 cycle ORAM latency/LLC miss; 25 KB moved per access
• Measure overall performance and on-chip power

• Each experiment runs for 200-250 billion instructions

• We compare against:
  – Insecure systems (no ORAM)
  – ORAM w/o timing protection
  – ORAM w/ various static rates

• Sweep knobs \( E, R \)

• First epoch = \( 2^{30} \) cycles
• Max runtime = \( 2^{62} \) cycles (>100 years)

For each \( R \): set of \( R \) rate candidates same for all benchmarks

Leakage = \# epochs \( \times \log_2(R) \)

\( \# \text{ epochs} = \frac{62 - 30}{\log_2(E)} \)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

Avg perf overhead (vs. no security)

Avg pwr overhead (vs. no security)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- Avg perf overhead (vs. no security)
  - $R = 16$
  - $4x$

- Avg pwr overhead (vs. no security)
  - $5x$
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- Avg perf overhead (vs. no security)

- Avg pwr overhead (vs. no security)

$R = 16\ 8$

$4x$ $5x$
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

Avg perf overhead (vs. no security)

Avg pwr overhead (vs. no security)

$R = 16, 8, 4$

$4x$ $5x$
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- $R = 4 \rightarrow 2$, power increases by 60% (leakage drops by 2x)

Avg perf overhead (vs. no security)

Avg pwr overhead (vs. no security)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- Avg perf overhead (vs. no security)
  - $R = 16$, 8, 4, 2
  - $4x$

- Avg pwr overhead (vs. no security)
  - $R = 16$, 8, 4, 2
  - $8x$
  - $5x$

Fix $R = 4$, vary $E$

- $E = \_\_\_$

• $R = 4 \rightarrow 2$, power increases by 60%
  (leakage drops by 2x)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- Avg perf overhead (vs. no security)
- Avg pwr overhead (vs. no security)

$R = 16, 8, 4, 2$

Fix $R = 4$, vary $E$

$E = 2$

- $R = 4 \rightarrow 2$, power increases by 60%

(leakage drops by 2x)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

- Avg perf overhead (vs. no security)
- Avg pwr overhead (vs. no security)

$R = 16 \ 8 \ 4 \ 2$

- $R = 4 \rightarrow 2$, power increases by 60%
- (leakage drops by 2x)

Fix $R = 4$, vary $E$

$E = 2 \ 4$
Trading off efficiency ↔ leakage

- **Fix E = 2, vary R**
  - Avg perf overhead (vs. no security)
    - R = 16 8 4 2
  - Avg pwr overhead (vs. no security)
    - R = 16 8 4 2

- **Fix R = 4, vary E**
  - Avg perf overhead (vs. no security)
    - E = 2 4 8
  - Avg pwr overhead (vs. no security)
    - E = 2 4 8

- **R = 4 → 2**, power increases by **60%** (leakage drops by **2x**)
Trading off efficiency ↔ leakage

Fix $E = 2$, vary $R$

Avg perf overhead (vs. no security)

- $R = 16$: 4x
- $R = 8$: 8x
- $R = 4$: 5x
- $R = 2$: 8x

Avg pwr overhead (vs. no security)

- $R = 16$: 4x
- $R = 8$: 8x
- $R = 4$: 5x
- $R = 2$: 5x

Fix $R = 4$, vary $E$

- $E = 2$: 8x
- $E = 4$: 5x
- $E = 8$: 8x
- $E = 16$: 5x

• $R = 4 \rightarrow 2$, power increases by 60% (leakage drops by 2x)
• $E = 4 \rightarrow 16$, performance drops by 5% (leakage drops by 2x)
Comparison to insecure and static

\[ R = 4, \ E = 4 \] point (favors performance) relative to …
Comparison to insecure and static

\[ R = 4, \; E = 4 \] point (favors performance) relative to …

<table>
<thead>
<tr>
<th>ORAM – no timing protection</th>
<th>Perf gain</th>
<th>Pwr efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-20%</td>
<td>-12%</td>
</tr>
</tbody>
</table>
Comparison to insecure and static

R = 4, E = 4 point (favors performance) relative to …

<table>
<thead>
<tr>
<th></th>
<th>Perf gain</th>
<th>Pwr efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORAM – no timing protection</td>
<td>-20%</td>
<td>-12%</td>
</tr>
<tr>
<td>Static – fast</td>
<td>Even</td>
<td>34%</td>
</tr>
</tbody>
</table>
Comparison to insecure and static

R = 4, E = 4 point (favors performance) relative to …

<table>
<thead>
<tr>
<th></th>
<th>Perf gain</th>
<th>Pwr efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORAM – no timing protection</td>
<td>-20%</td>
<td>-12%</td>
</tr>
<tr>
<td>Static – fast</td>
<td>Even</td>
<td>34%</td>
</tr>
<tr>
<td>Static – slow</td>
<td>30%</td>
<td>Even</td>
</tr>
</tbody>
</table>
Comparison to insecure and static

R = 4, E = 4 point (favors performance) relative to ...

<table>
<thead>
<tr>
<th></th>
<th>Perf gain</th>
<th>Pwr efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORAM – no timing protection</td>
<td>-20%</td>
<td>-12%</td>
</tr>
<tr>
<td>Static – fast</td>
<td>Even</td>
<td>34%</td>
</tr>
<tr>
<td>Static – slow</td>
<td>30%</td>
<td>Even</td>
</tr>
</tbody>
</table>

Cost (R = 4, E = 4): ≤ 32 bits leaked
Leakage aware processors use knobs (e.g., E, R) that control leakage & trade-off that leakage for efficiency

• Big idea: leakage across channels is additive!
  – $T_1 \times T_2$ traces = $\log_2(T_1) + \log_2(T_2)$ leakage
  – Good leakage scalability to other timing channels (e.g., cache)
  – More general usage models for encrypted computation
Thank you!

Questions?
• **SVF**

• **SurfNoc**

• **RP/PL $**

• **No-Mo $**

• **Ren et al.**

• **Phantom**
Citations (2)

• Smith et al.

• Kopf et al.

• Askarov et al.
Backup slides
Other topics in paper

- Preventing replay attacks
  - Leak \( L \) bits per execution, \( N \) replays with \( N \) different programs
    \[ = L \times N \text{ bits leaked} \]

- Performance as epochs become sparser

- Extending work to different channels (e.g., shared $)$
Rate learners

- **Algorithm:**
  - Calculate NewRate = Time / (# real ORAM reqs) [avg offered load]
  - Map NewRate to one of the $R$ candidate rates

- **Mechanisms in place to compensate for various effects**
**Rate learners**

- **Algorithm:**
  - Calculate $\text{NewRate} = \text{Time} / (\# \text{real ORAM reqs})$ [avg offered load]
  - Map $\text{NewRate}$ to one of the $R$ candidate rates

- **Mechanisms in place to compensate for various effects**

**Example:**
NewRate depends on last epoch’s rate

- **Observable rate**
- **“Real” rate**

Diagram:
- Time axis
- Epoch transition
- Slower rate transitions
Rate learners

• **Algorithm:**
  – Calculate $\text{NewRate} = \text{Time} / (\# \text{real ORAM reqs})$ [avg offered load]
  – Map NewRate to one of the $R$ candidate rates

• **Mechanisms in place to compensate for various effects**

**Example:**
NewRate depends on last epoch’s rate

- Observable rate
- “Real” rate

Epoch transition

Diagram showing time and rate transitions.
Rate learners

• **Algorithm:**
  - Calculate $\text{NewRate} = \text{Time} / (\# \text{real ORAM reqs})$ [avg offered load]
  - Map $\text{NewRate}$ to one of the $R$ candidate rates

• **Mechanisms in place to compensate for various effects**

**Example:**
NewRate depends on last epoch’s rate

Observable rate

“Real” rate

---

Epoch transition

Time
Rate learners

- **Algorithm:**
  - Calculate \( \text{NewRate} = \frac{\text{Time}}{\# \text{real ORAM reqs}} \) [avg offered load]
  - Map \( \text{NewRate} \) to one of the \( R \) candidate rates

- **Mechanisms in place to compensate for various effects**

**Example:**

NewRate depends on last epoch’s rate.

Observable rate

“Real” rate

\[\text{slower} \quad \text{slower}\]

Epoch transition

\[\text{Time}\]
**Rate learners**

- **Algorithm:**
  - Calculate $\text{NewRate} = \frac{\text{Time}}{\text{(\# real ORAM reqs)}}$ [avg offered load]
  - Map NewRate to one of the $R$ candidate rates

- **Mechanisms in place to compensate for various effects**

**Example:**

NewRate depends on last epoch’s rate

Observable rate

“Real” rate

Epoch transition
Rate learners

- **Algorithm:**
  - Calculate $\text{NewRate} = \frac{\text{Time}}{\text{(# real ORAM reqs)}}$ [avg offered load]
  - Map NewRate to one of the $R$ candidate rates

- **Mechanisms in place to compensate for various effects**

**Example:**
NewRate depends on last epoch’s rate

**What we want**
Rate learners

• **Algorithm:**
  - Calculate $\text{NewRate} = \text{Time} / (\# \text{real ORAM reqs})$ [avg offered load]
  - Map $\text{NewRate}$ to one of the $R$ candidate rates

• **Mechanisms in place to compensate for various effects**

**Example:**
NewRate depends on last epoch’s rate

<table>
<thead>
<tr>
<th>Observable rate</th>
<th>“Real” rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>slimmer</td>
<td>slimmer</td>
</tr>
</tbody>
</table>

Epoch transition

• **End result …**
  - Small circuit
    * Performance counters
    * Subtracts, constant shifts
  - Self contained
    * Snoops off the LLC→ORAM controller queue
Performance over time: H264

- **Notation:** Vertical bars mark epoch transitions
Performance over time: H264

Becomes memory bound

- **Notation:** Vertical bars mark epoch transitions
Performance over time: H264

- **Notation:** Vertical bars mark epoch transitions
- **Trade-offs explained in paper in detail**
- **Static scheme requires ~4X power!**