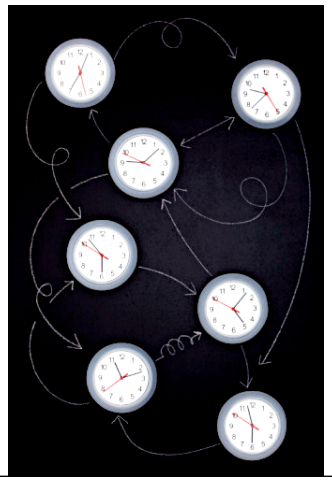
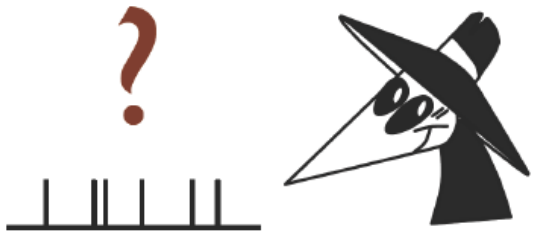
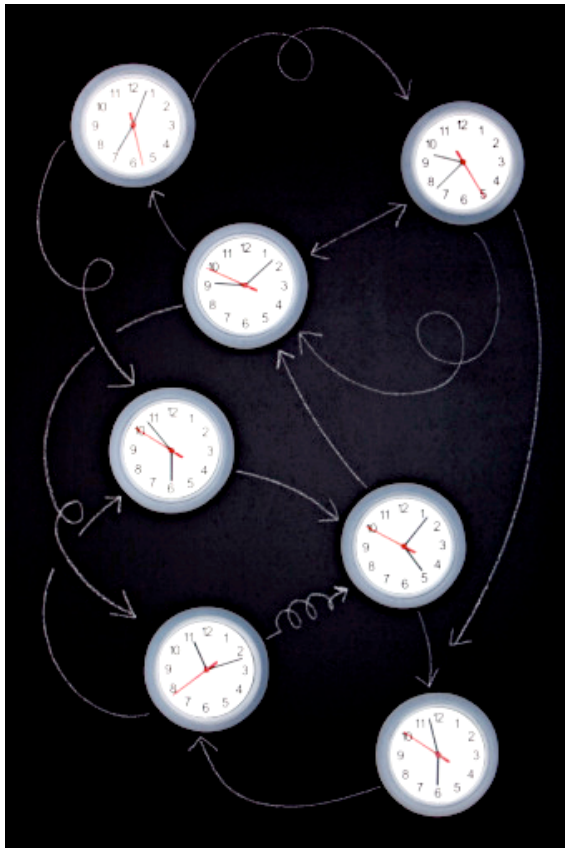
New methods for controlling timing channels

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

+

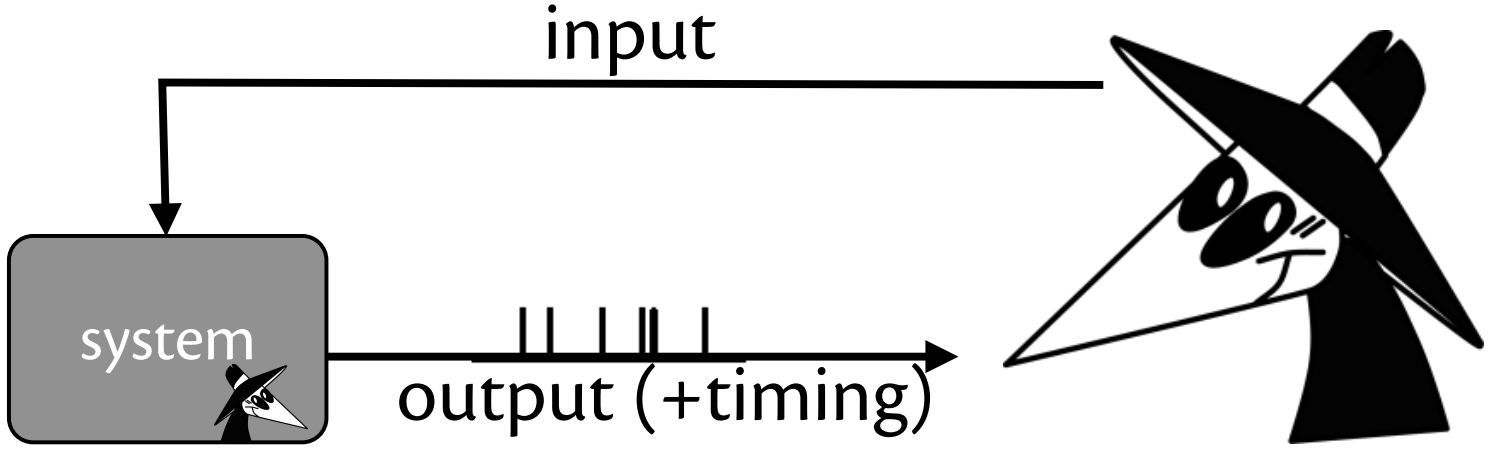
lack of feasible defenses

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,…)

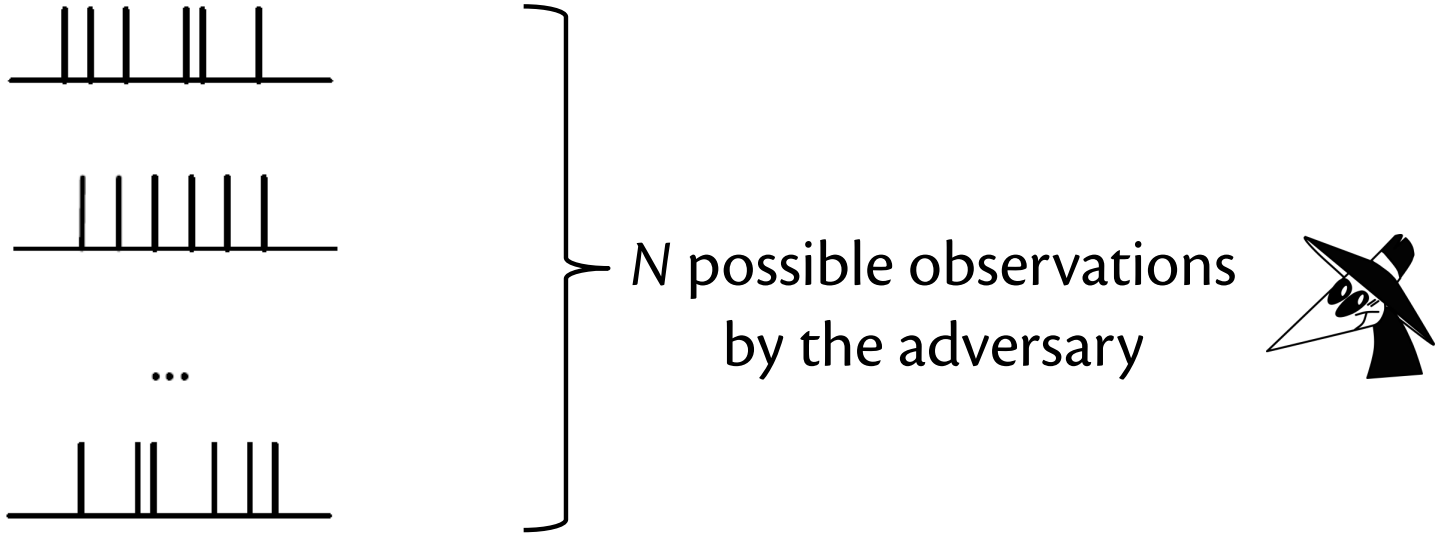


Timing channel mitigation

- 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 → leakage



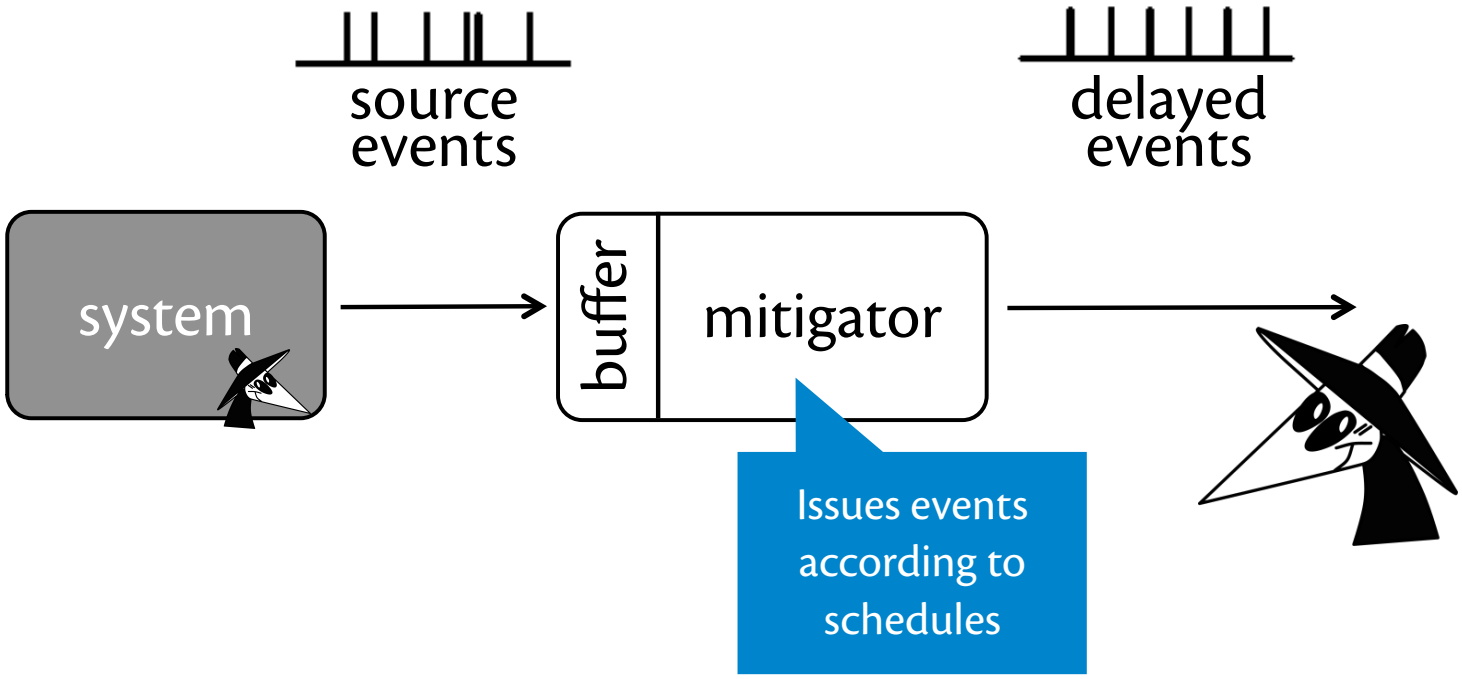
Leakage in bits = $\log_2 N$

A bound on:

mutual information (Shannon entropy)

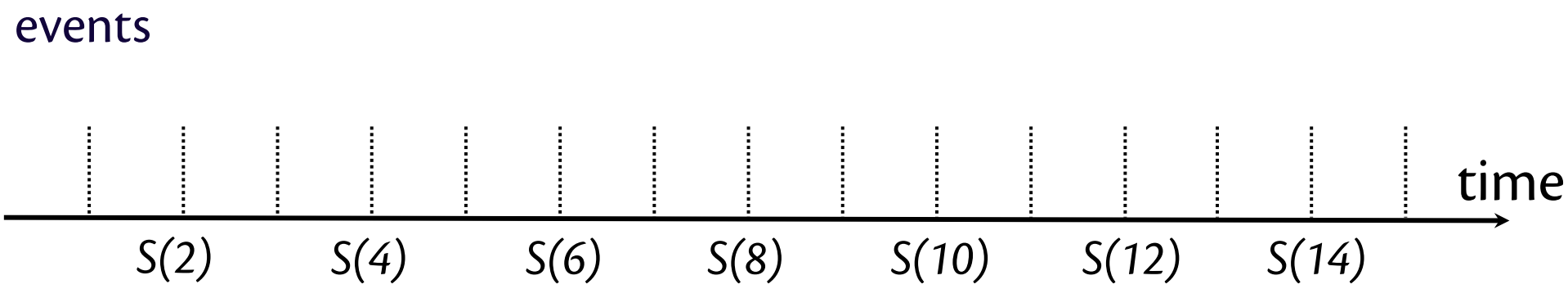
min-entropy

Black-box predictive mitigation



Prediction by doubling

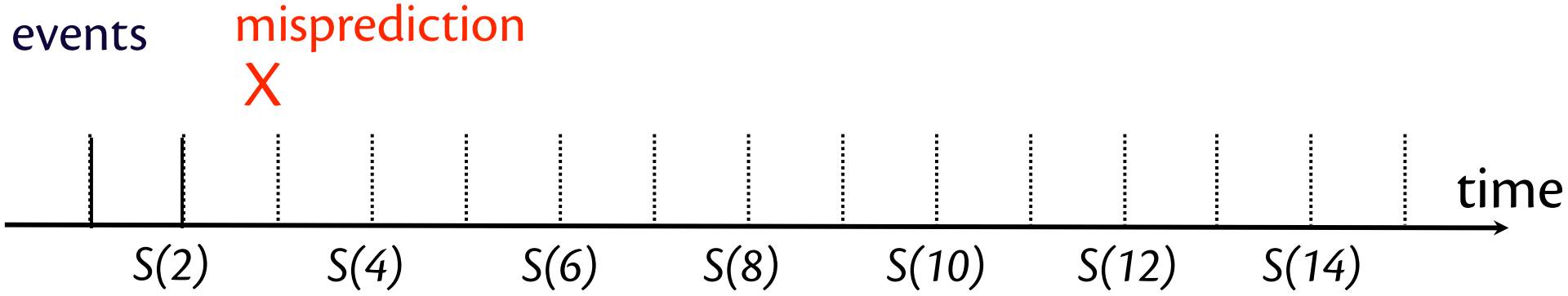
predictions: when mitigator expects to deliver events



Mitigator starts with a fixed schedule *S*

S(*n*) – prediction for *n*th event

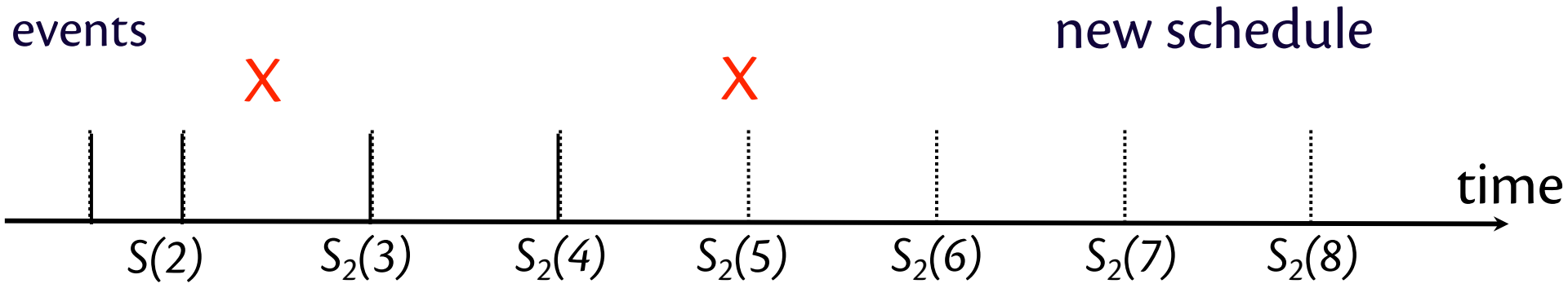
Example: Doubling



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

little information leaked

Example: Doubling

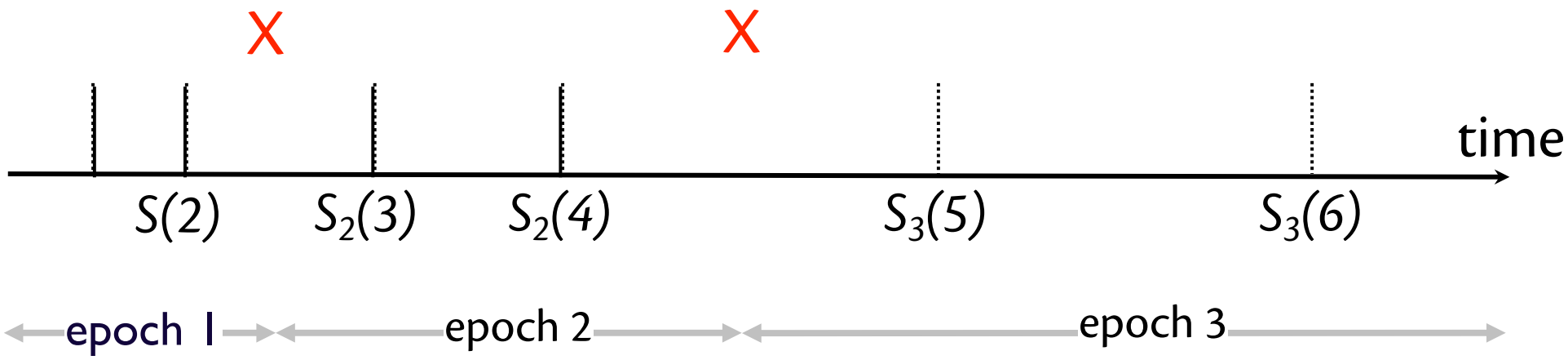


Adversary observes mispredictions ⇒ **information leaked**

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? $(M+1)^N$

events

Depends on prediction scheme

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

- Leakage with doubling scheme:

$N = O(\log T)$

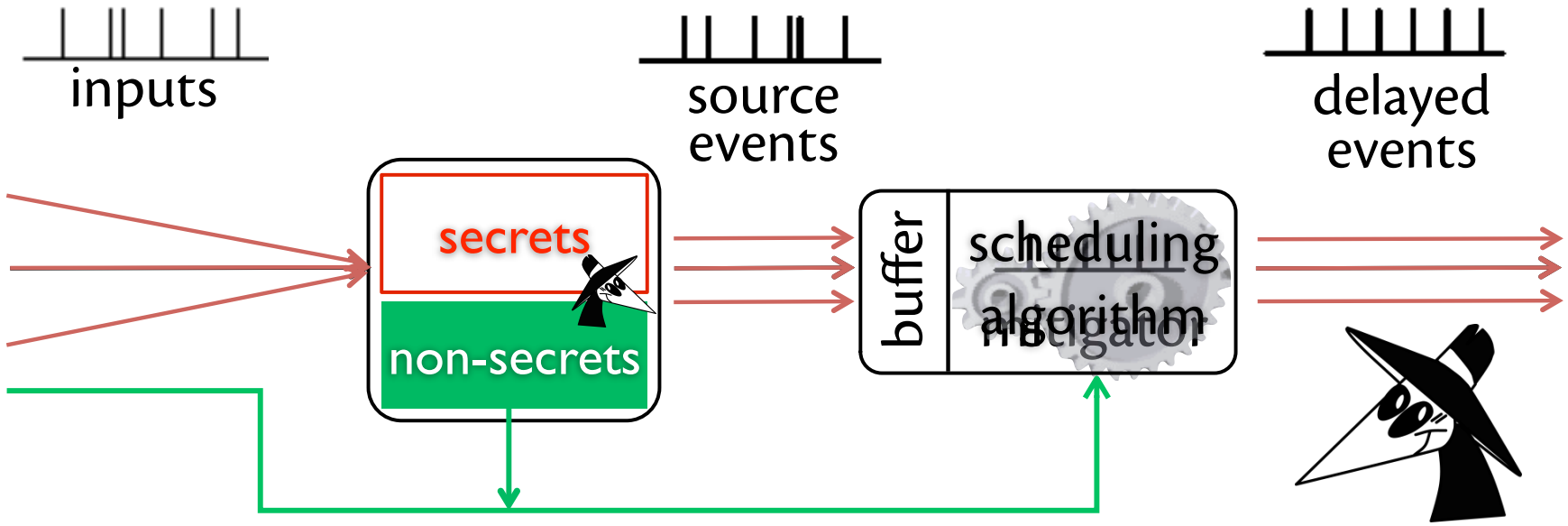
leakage $\leq O(\log^2 T)$

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

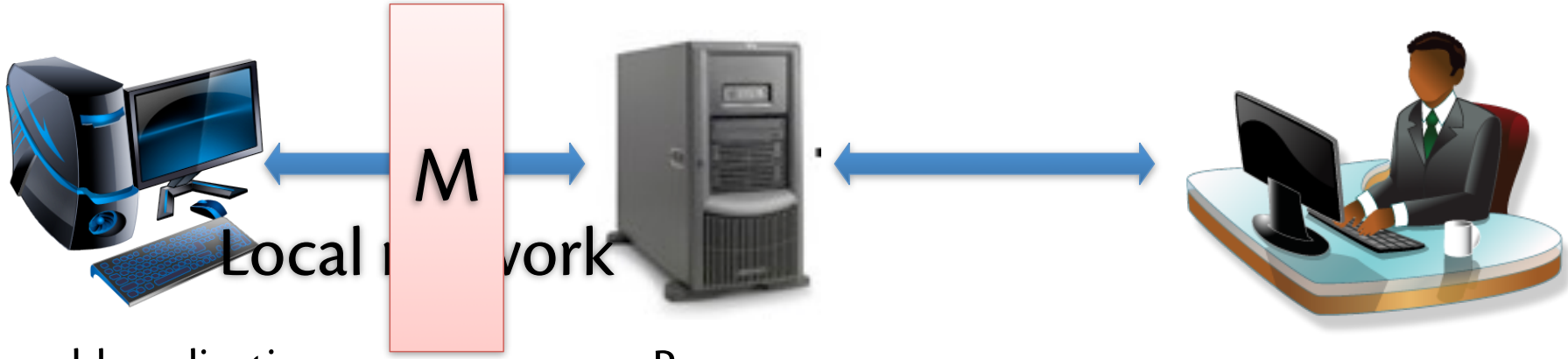
- Public payloads in requests, such as URLs

  www.example.com/index.html vs.
  www.example.com/background.gif

- Time of input request

Evaluation

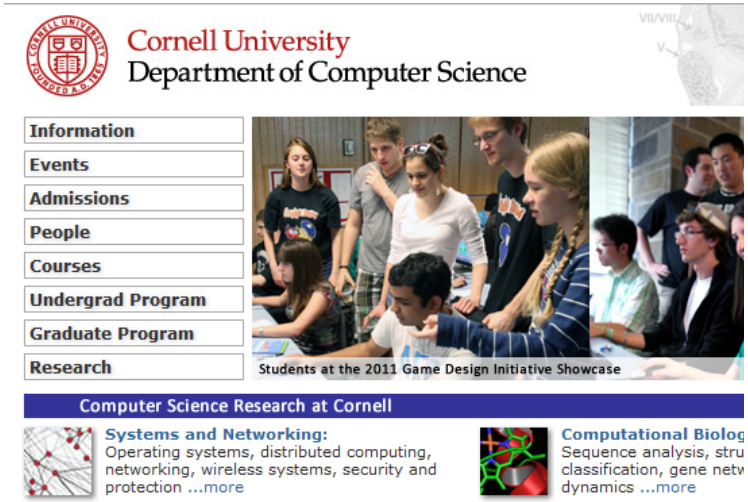
Real-world web applications (with HTTP(S) proxy)

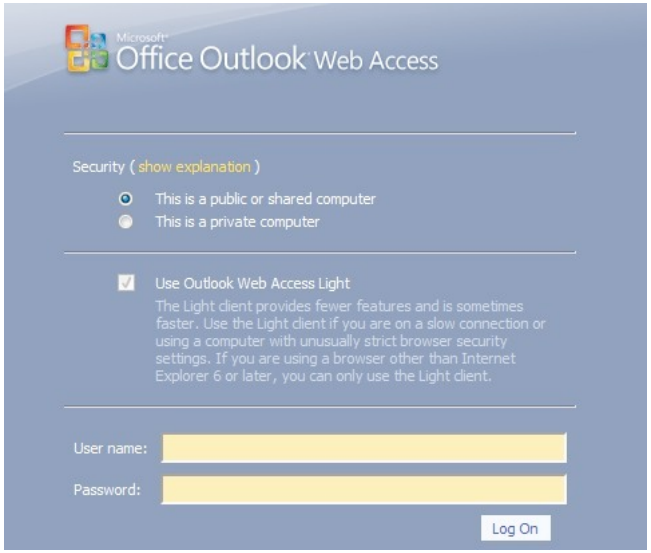


Real-world applications

Proxy

Client





Mitigating Web proxy

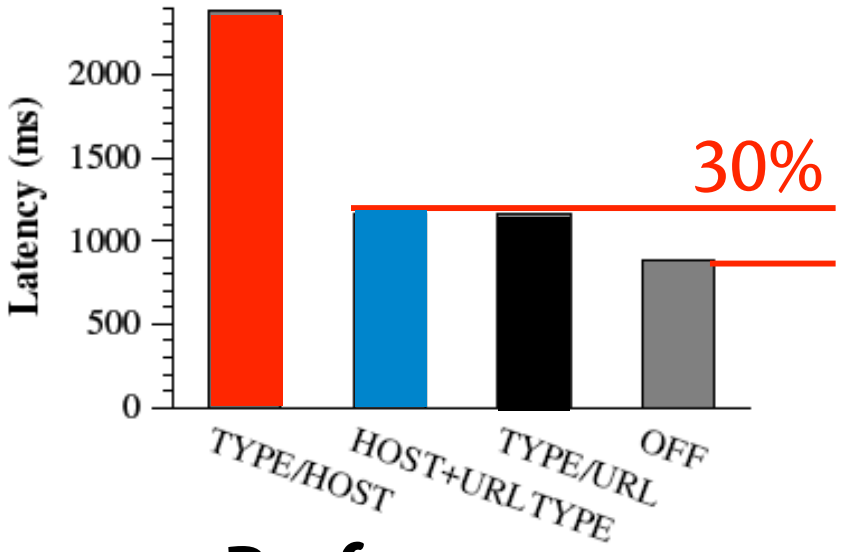
Demo

Experiments with Web applications

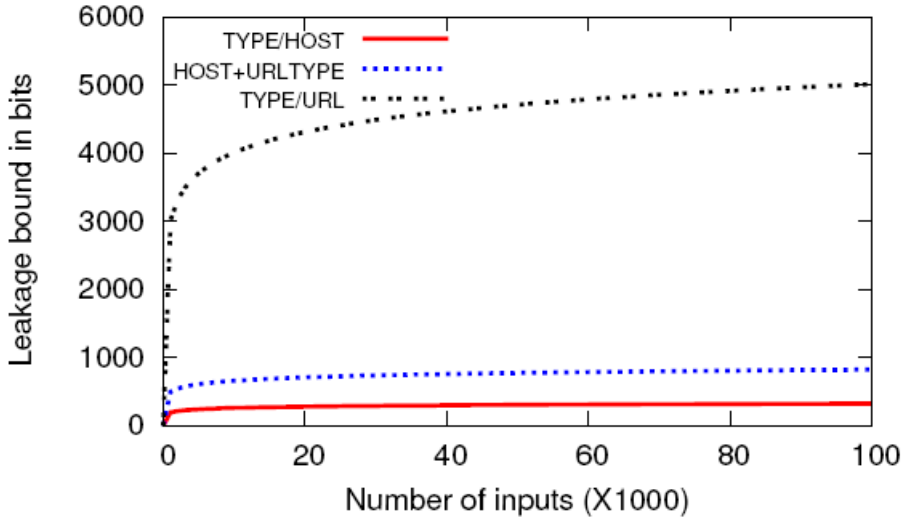
Mitigating department homepage via HTTP

(**49** different requests)

- Different prediction schemes trade off security vs. performance. With HOST+URLTYPE scheme:
  - **~30%** latency overhead
  - < **850bits** for 100,000 inputs



Performance

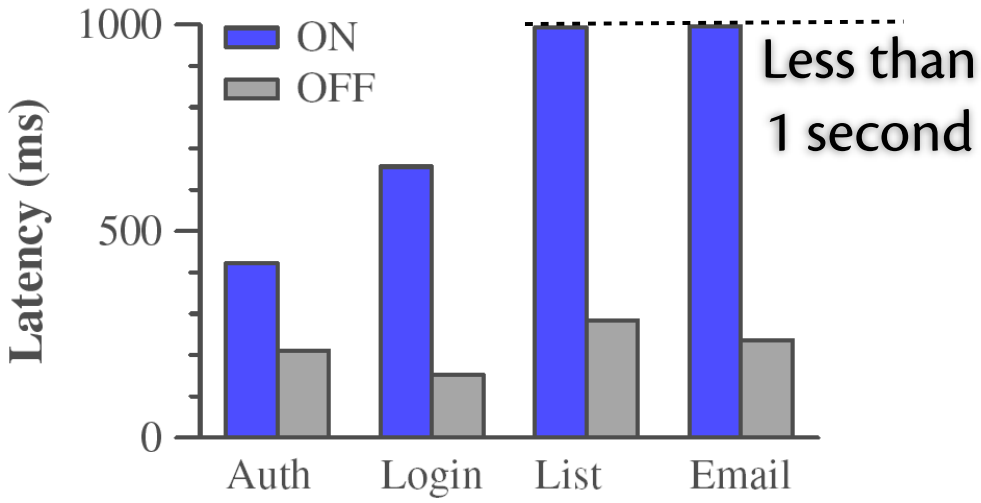


Security

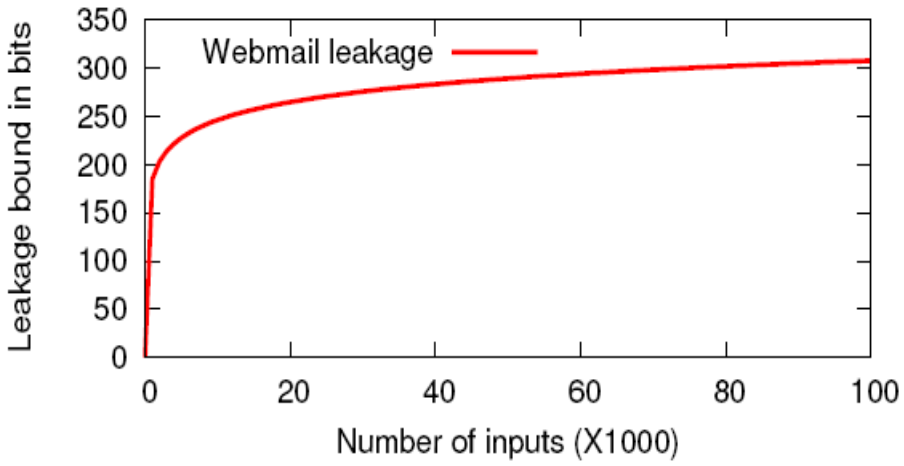
Experiments with Web applications

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

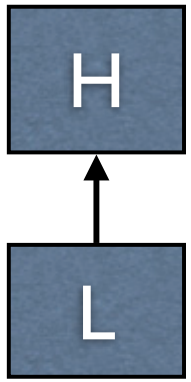
- Timing mitigation for cryptographic operations
  [Kocher 96, Kopf & Durmuth 09, Kopf & Smith 10]
  - Assumes input blinding
- NRL Pump/Network Pump [Kang et. al. 93, 96]
  - Addresses covert channels from input acks
  - Linear bound
- Information theory community [Hu 91, Giles&Hajek 02]
  - Timing 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

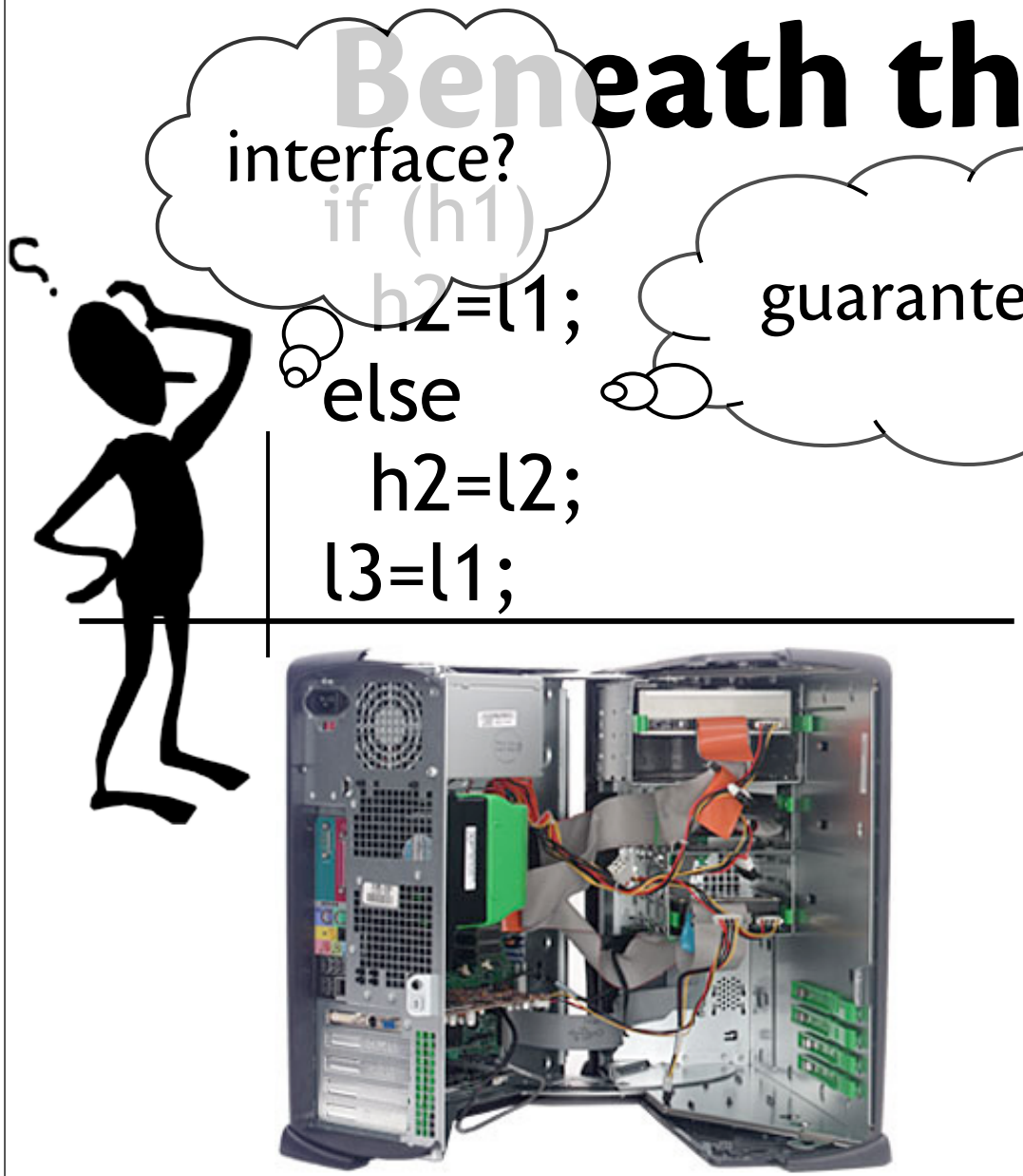
```
if (h)
  sleep(1);
else
  sleep(2);
```

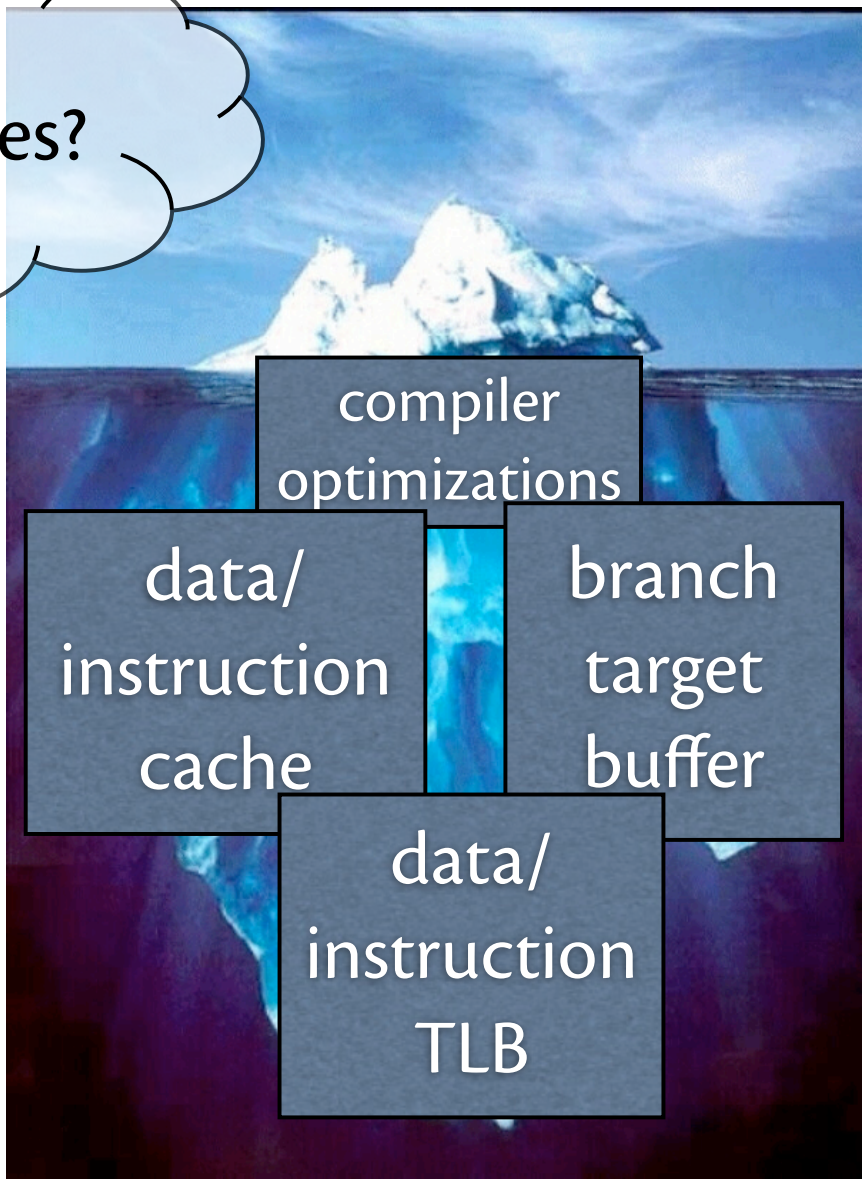
A subtle example

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

Data cache affects timing!

Beneath the surface





A language-level abstraction

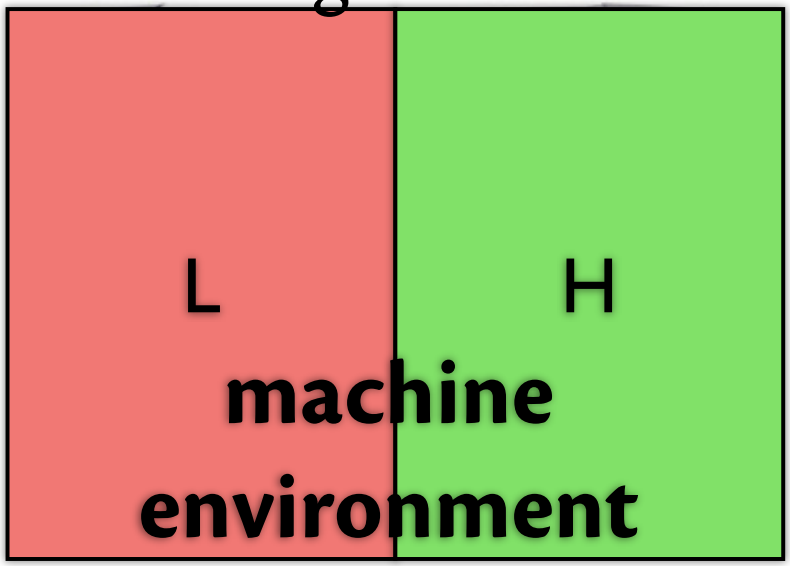
- Each operation has **read label**, **write label** governing interaction with machine environment

$(x := e)$

$[\ell_r, \ell_w]$

Machine environment: state affecting timing but invisible at language level

machine env. logically partitioned by security level (e.g. high cache vs. low cache)



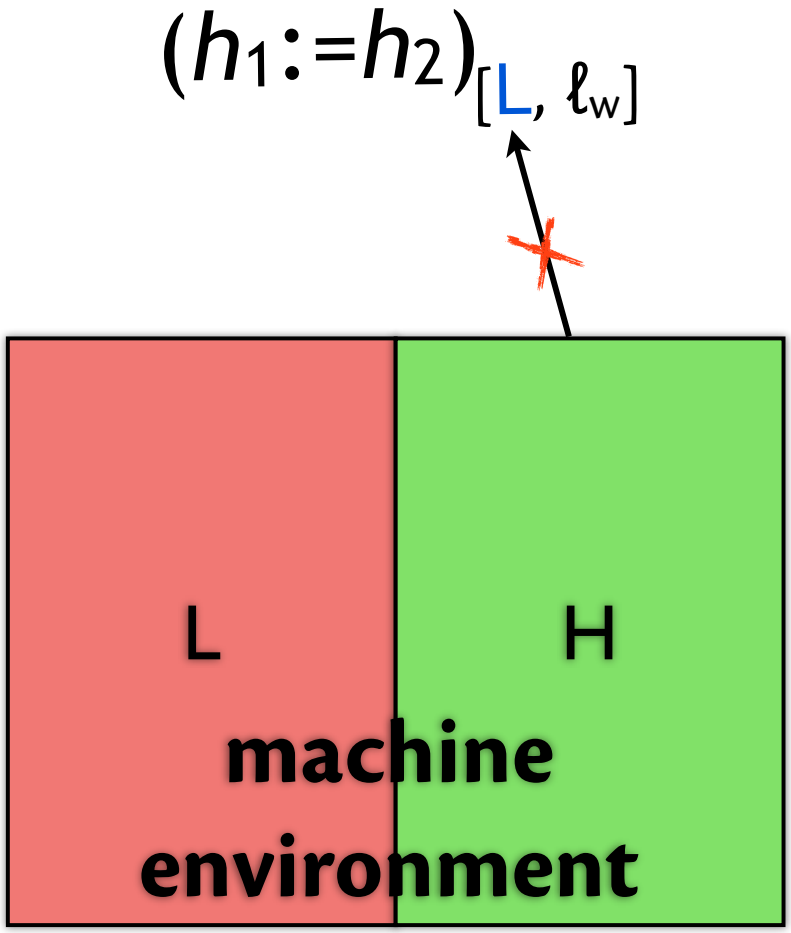
Does *not* include language-visible state (memory)

Read label

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

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

= upper bound on influence

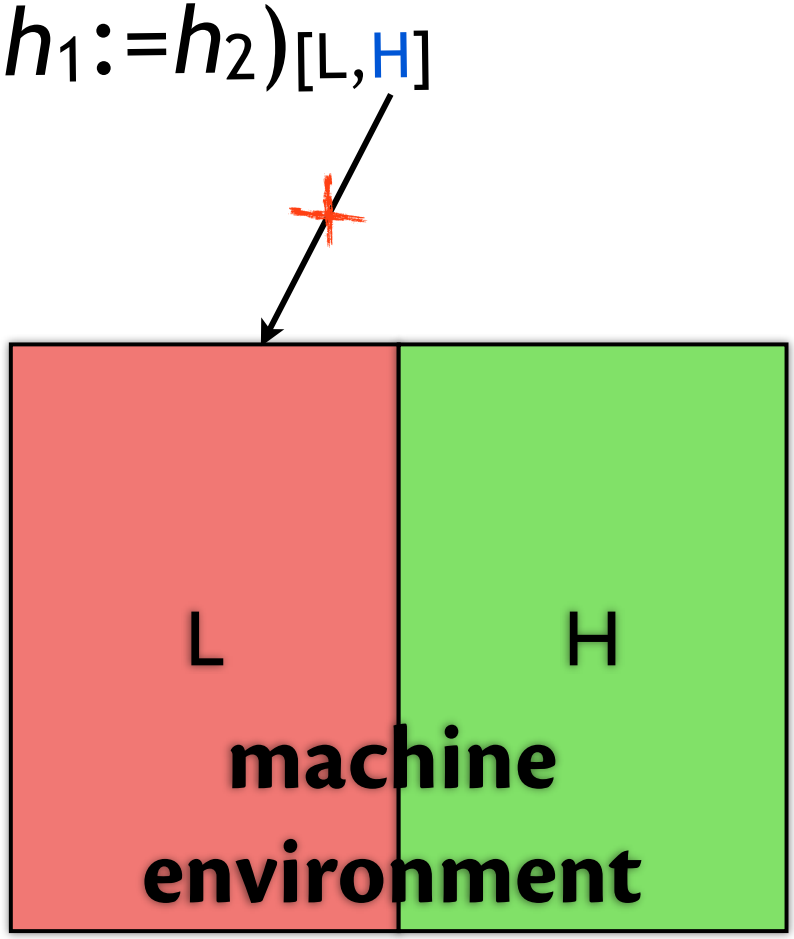


Write label

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

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

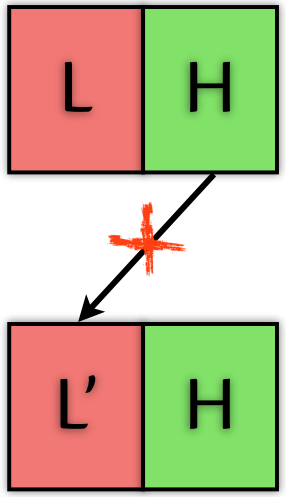
= *lower* bound on effects



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 ⇒ time to run c depends on information at (at most) label T

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

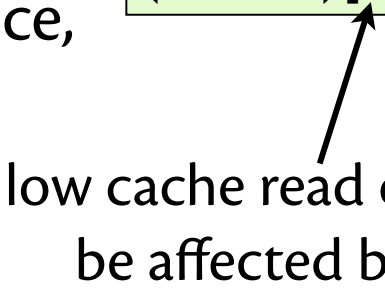
Examples:

$c_{[H,\ell w]}$: H

sleep(h) : H

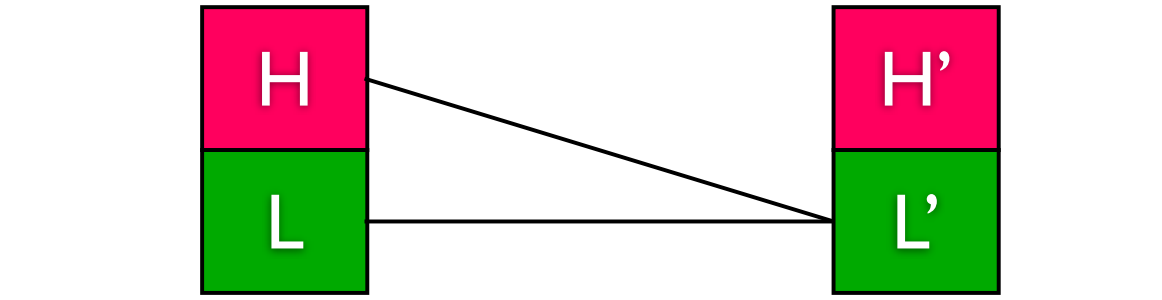
$(x := y)_{[L,L]}$: L

if (h_1)
  $(h_2 := l_1)_{[L,H]}$;
else
  $(h_2 := l_2)_{[L,H]}$;
$(l_3 := l_1)_{[L,L]}$

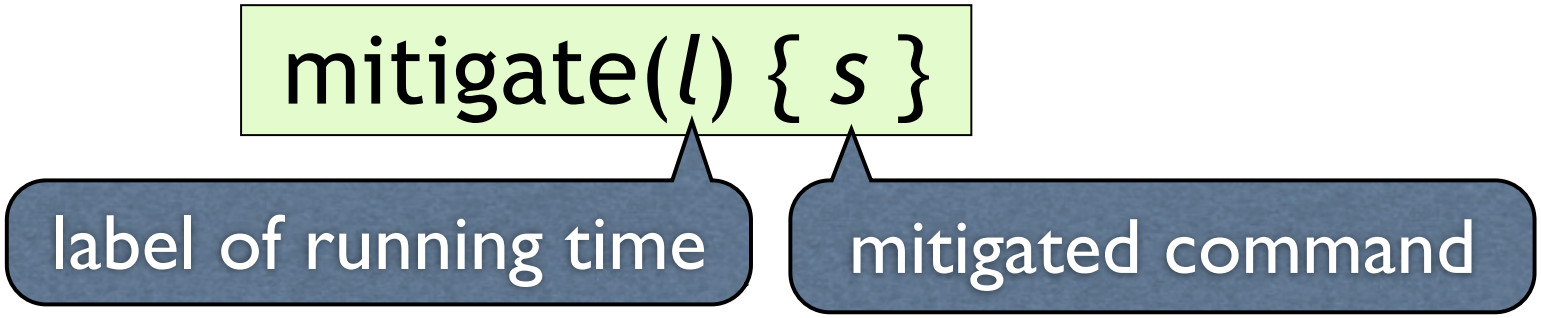


Formal results

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



Language-level mitigation



- Executes *s* but adds time using predictive mitigation
  - New expressive power:
    sleep(h) : H *but* mitigate(l) { sleep (h) } : 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

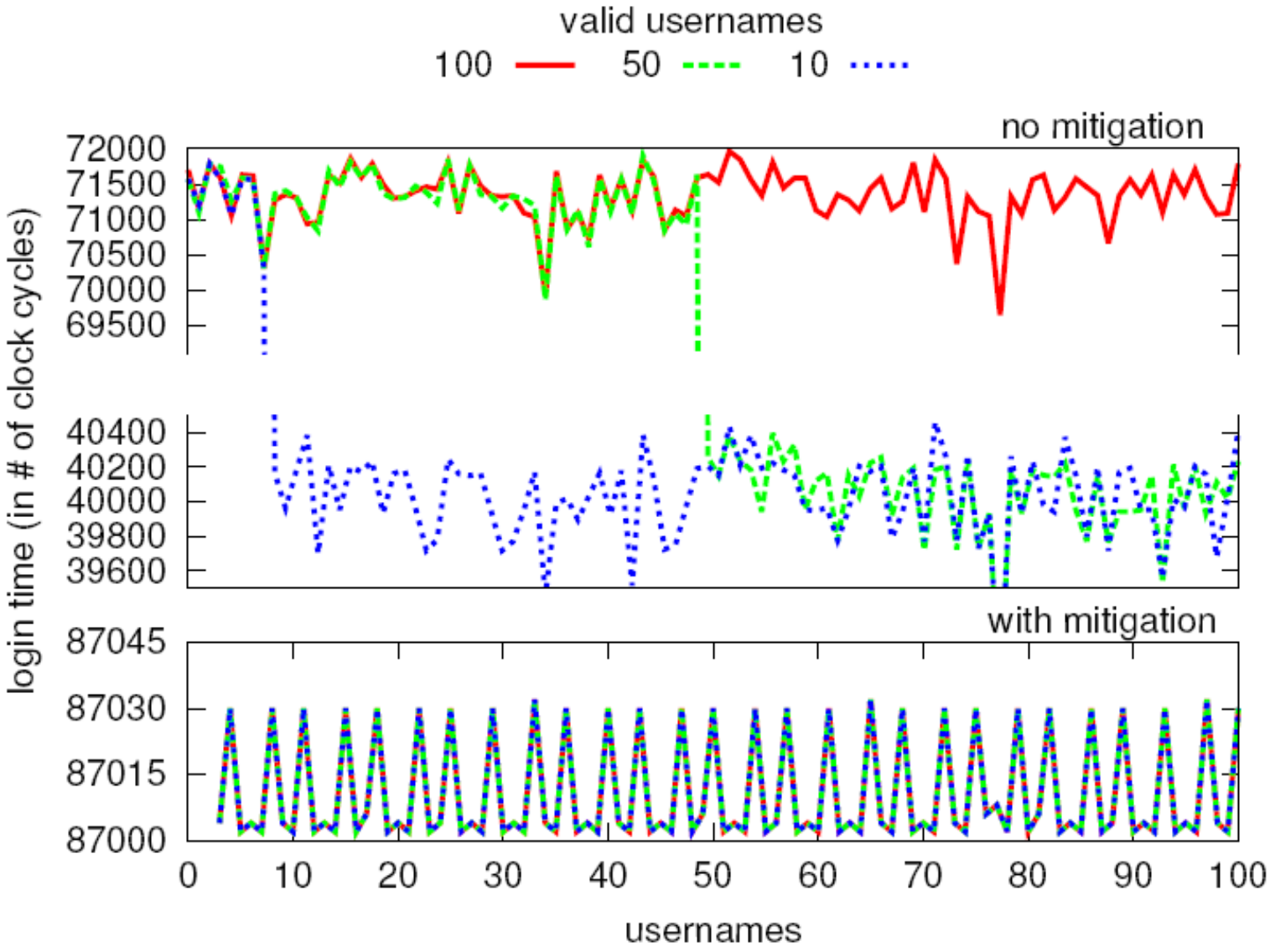
Name	# of sets	issue	block size	latency
L1 Data Cache	128	4-way	32 byte	1 cycle
L2 Data Cache	1024	4-way	64 byte	6 cycles
L1 Inst. Cache	512	1-way	32 byte	1 cycle
L2 Inst. Cache	1024	4-way	64 byte	6 cycles
Data TLB	16	4-way	4KB	30 cycles
Instruction TLB	32	4-way	4KB	30 cycles

- 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



Performance

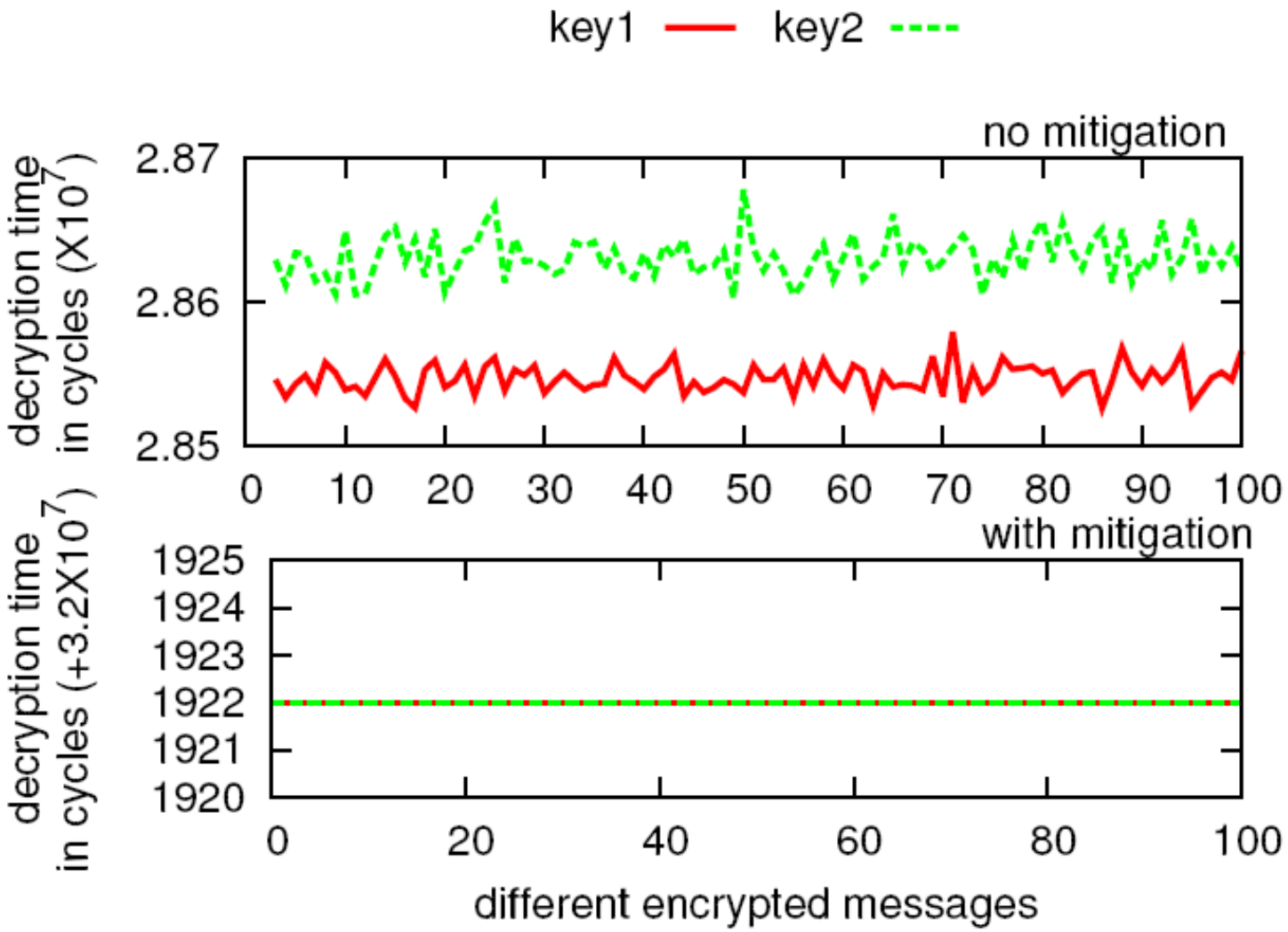
- nopar: unmodified hardware
- moff: secure hardware, no mitigation
- mon: secure hardware with mitigation

	nopar	moff	mon
ave. time (valid)	70618	78610	86132
ave. time (invalid)	39593	43756	86147
overhead (valid)	1	1.11	1.22

RSA

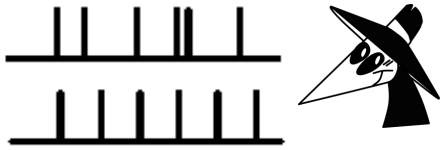
- RSA reference implementation
- Secret: private keys
- Inputs: different encrypted messages

RSA behavior



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.