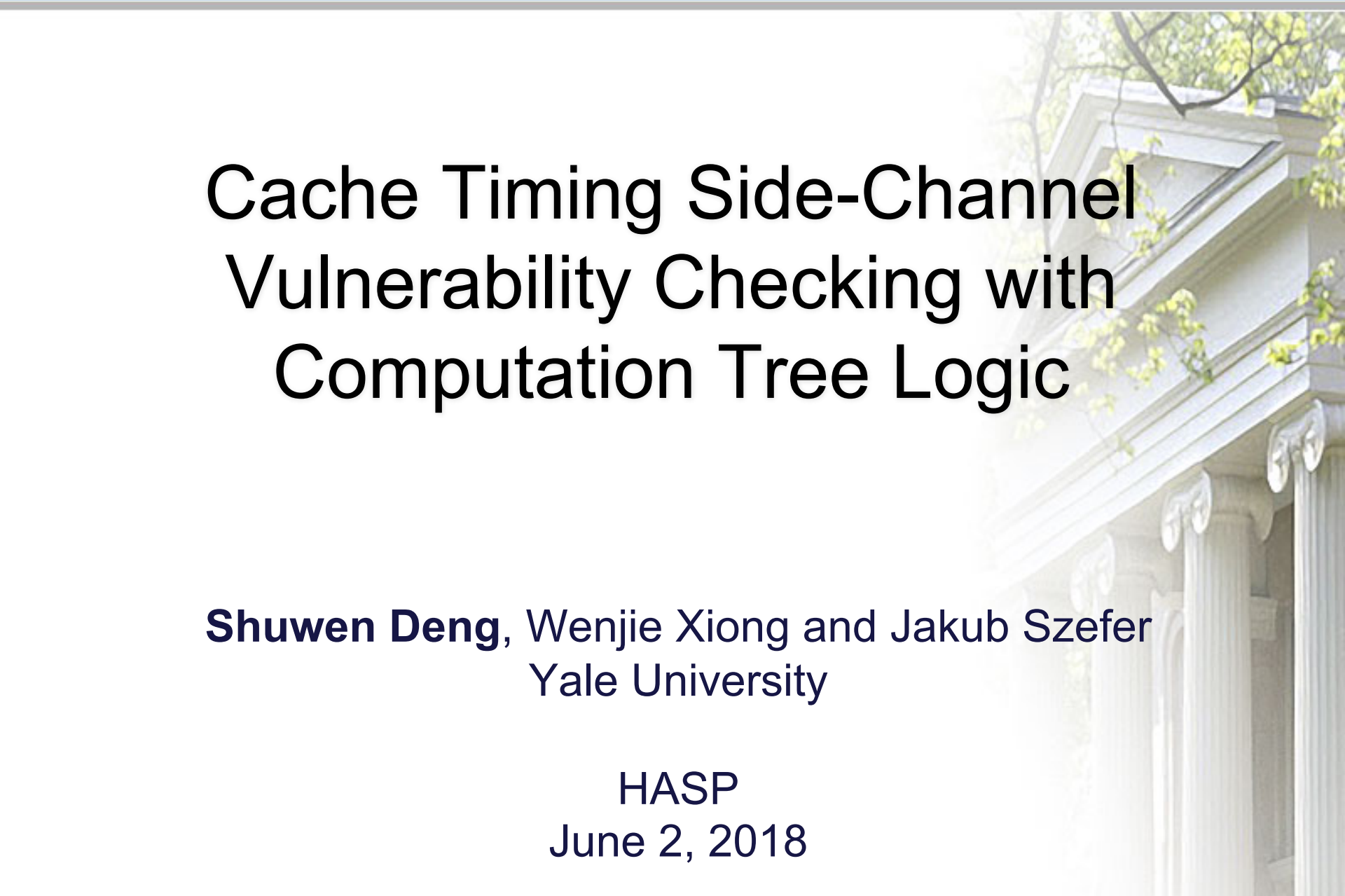
Yale



Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic

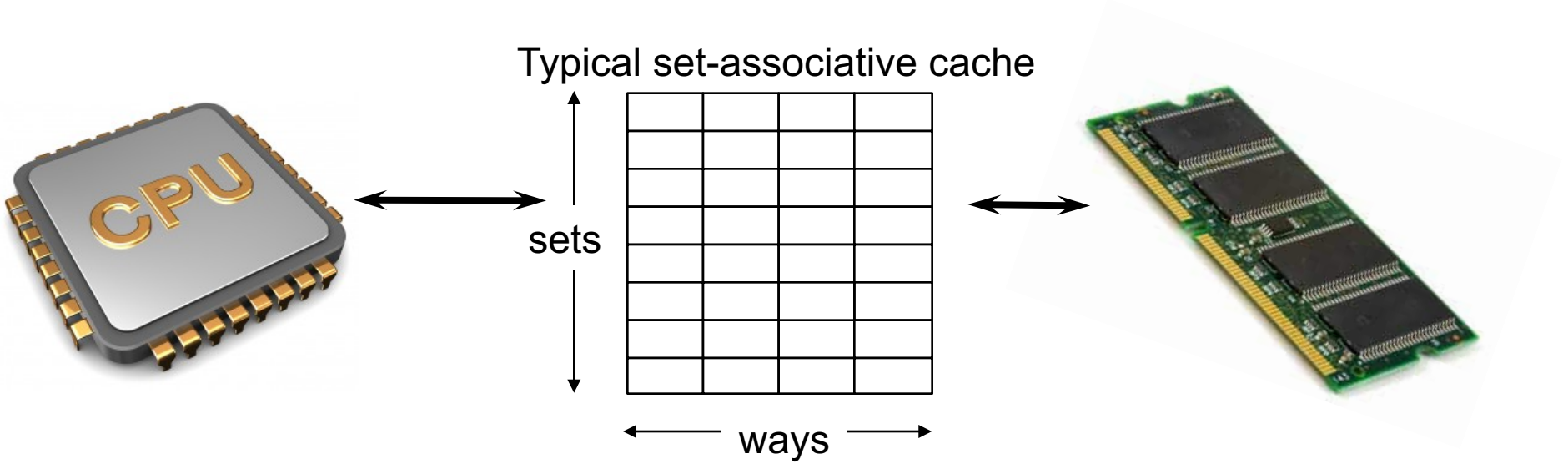
Shuwen Deng, Wenjie Xiong and Jakub Szefer
Yale University

HASP
June 2, 2018

Memory System

Yale

CPU Cache Memory



cache enables fast access to the data

Cache State Machine

Yale

timing latency ⟷
cache hit & miss

fast
access

slow
access

Cache Timing Side-Channel Attacks

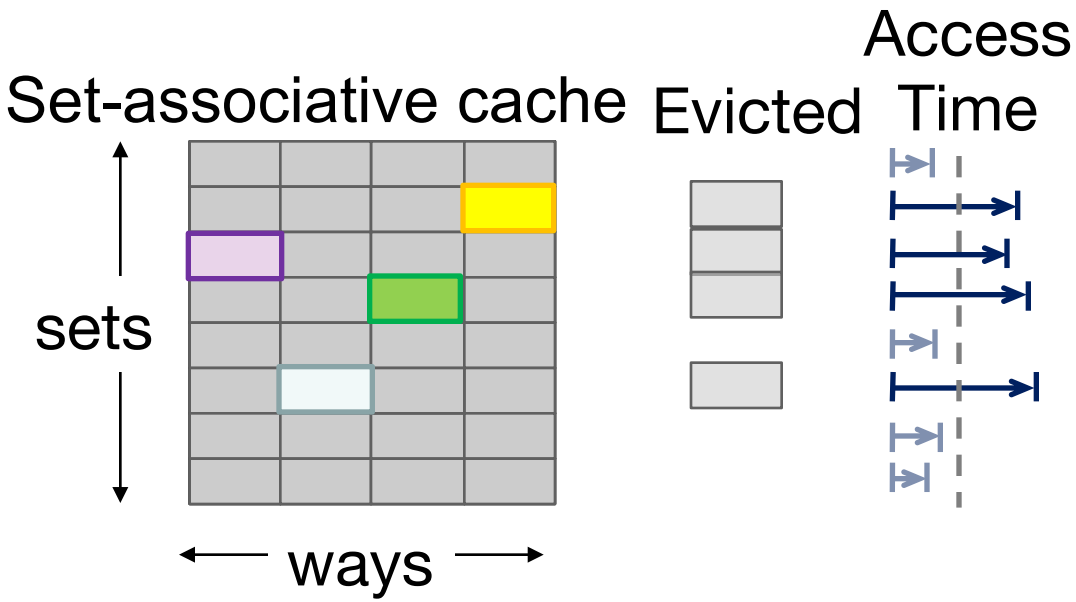
Yale

- For load/store instruction, time differs between hits and misses
- For flush instruction, time depends on data existence

- Attacker's Goal: get information of the address of victim's sensitive data by observing the timing difference
- Threat Model:
  - An attacker (A) shares the same cache with a victim (V)
  - The attacker cannot directly access the cache state machine
  - The attacker can observe the timing of the victim or itself
  - The attacker can combine timing observation with some other knowledge
    - The attacker knows some source code of the victim
    - The attacker can force victim to execute a specific function
- E.g. Flush + Reload Attack

E.g. Prime + Probe Attack

Yale



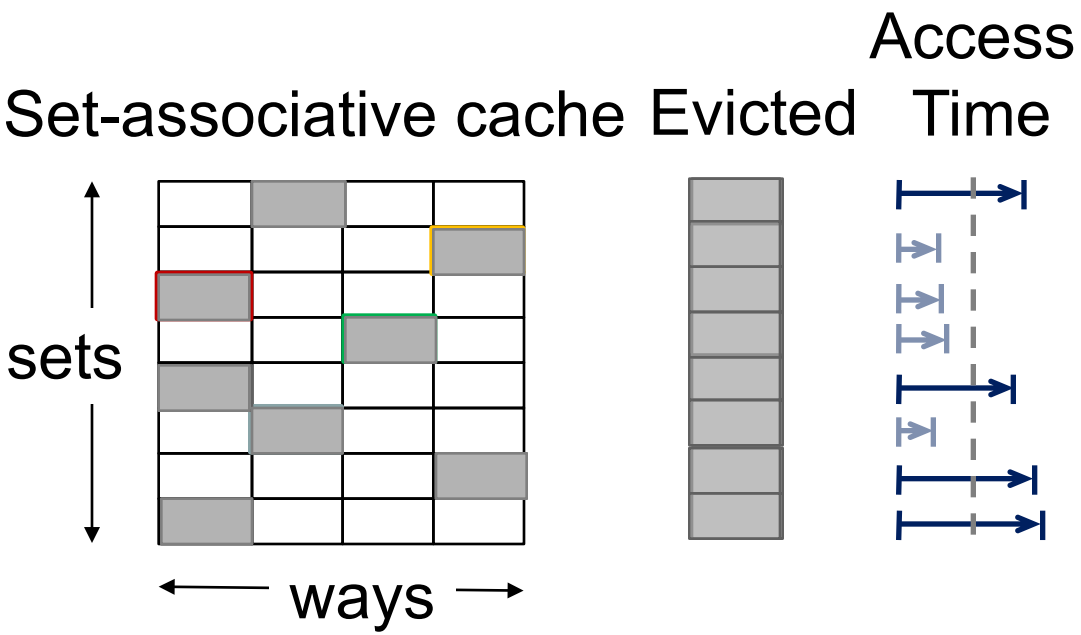
1- Attacker primes each cache set

2- Victim accesses critical data

3- Attacker probes each cache set (measure time)

E.g. Flush + Reload Attack

Yale



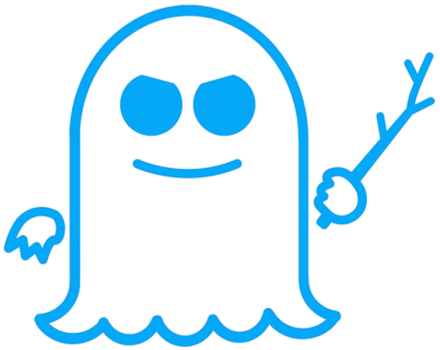
1- Flush each line in the cache

2- Victim accesses critical data

3- Attacker reloads critical data by running specific process (measure time)

Spectre & Meltdown Attack

Yale



- speculative executions
  - Variant 1: Bounds Check
  - Variant 2: Branch Target Injection
  - Variant 3: Rogue Data Cache
    - Variant 3a: Rogue System Register
  - Variant 4: Speculative Store
- timing side-channel in the cache

Spectre & Meltdown Attack

Yale

- Uses speculative executions
- Leverages timing side-channel in the cache

Contribution

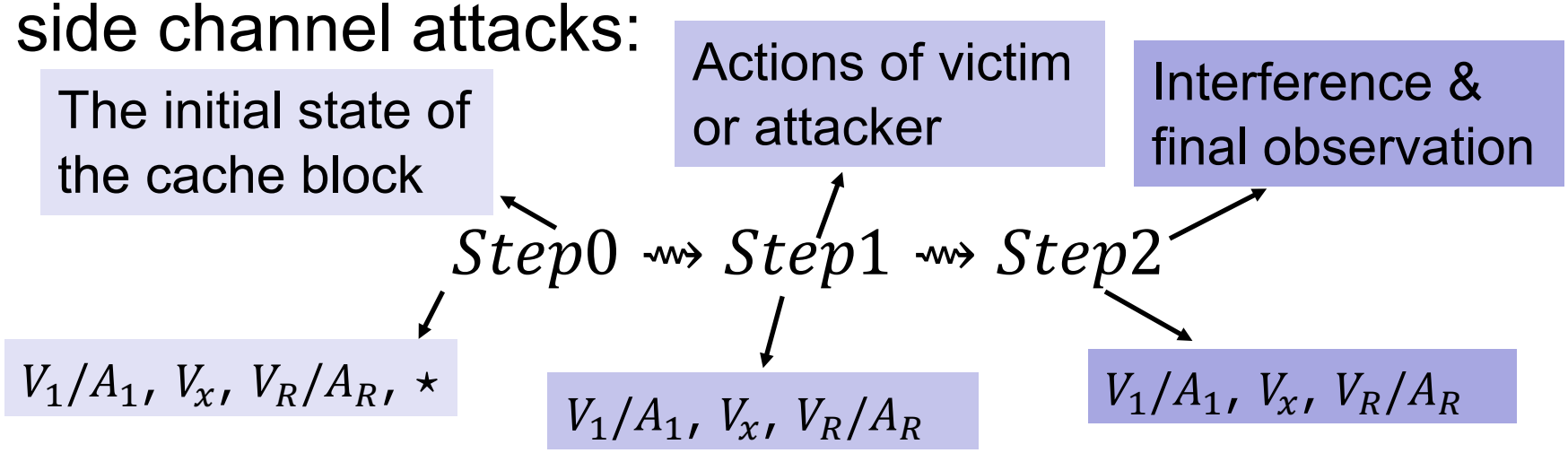
Yale

Develop Cache Access Model	Analyze Timing Vulnerabilities	Use Computation Tree Logic (CTL)
Three-step single-cache-block-access model construction	Exhaustive search for possible attacks based on three-step model	Model execution paths of the processor cache focusing on side-channel attacks

Three-Step Single-Cache-Block-Access Model

Yale

We use three steps to model all possible cache side channel attacks:

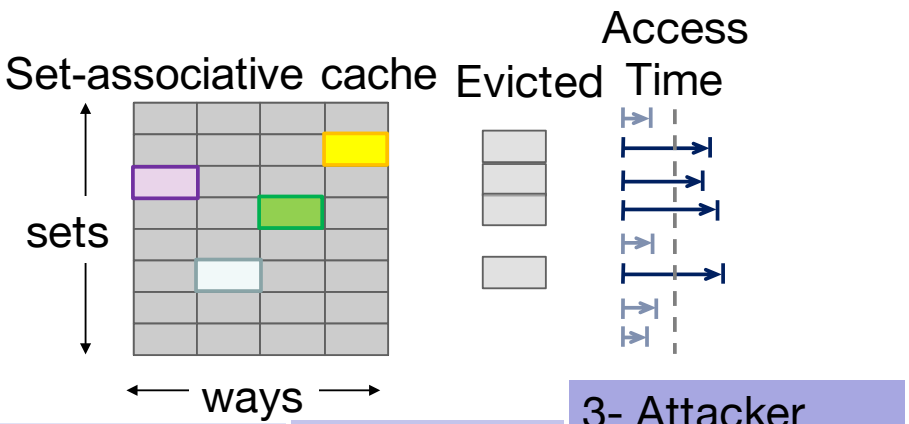


condition	description
V_1/A_1	A specific known memory location.
V_x	A piece of memory containing data from a range of victim's memory addresses is accessed.
V_R/A_R	single-cache-block access to "remove" the cache block contents
$\star$	Attacker has no knowledge about memory location

Vulnerability Examples

Yale

- Prime + Probe Attack

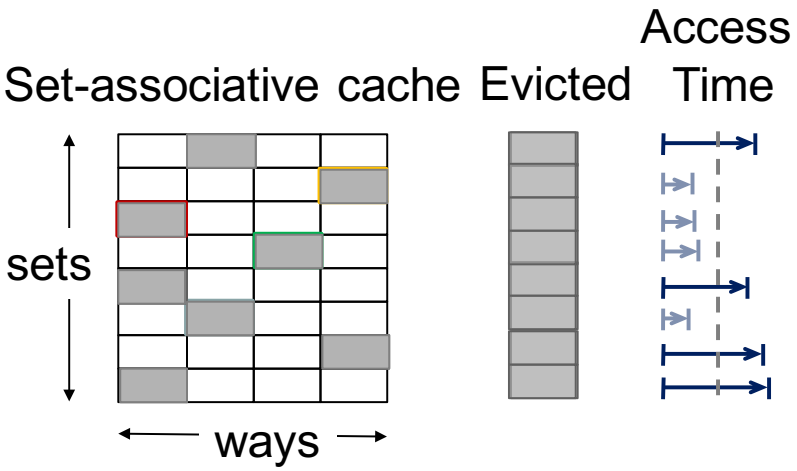


– $EF(E(E(A_1\ UV_x)UA_1))$

Vulnerability Examples

Yale

- Flush + Reload Attack



1- Flush each line in the cache

2- Victim accesses critical data

3- Attacker Reloads critical data by running specific process (measure time)

- $A_R \rightsquigarrow V_x \rightsquigarrow A_1$
- $V_R \rightsquigarrow V_x \rightsquigarrow A_1$

condition	description
V_1/A_1	A specific known memory location.
V_x	A piece of memory containing data from a range of victim's memory addresses is accessed.
V_R/A_R	single-cache-block access to "remove" the cache block contents

Soundness of Three-Step Model

Yale

Why three-step model can cover all?

- One cache access
  - Interference does not exist
- Two cache accesses
  - Same as three-step model with $Step0$ to be " $\star$ "
- More than three cache accesses
  - $\{\cdots \rightsquigarrow \star \rightsquigarrow \cdots\}$ can be divided into two parts
  - $\{\cdots \rightsquigarrow A_R \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow A_R \rightsquigarrow V_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow A_1 \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_x \rightsquigarrow V_x \rightsquigarrow \cdots\}$, ... can be reduced to $\{\cdots \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_x \rightsquigarrow \cdots\}$, ..., respectively
  - $\{\cdots \rightsquigarrow (A_R/V_R \ / \ A_1 \ / \ V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R \ / \ A_1 \ / \ V_1) \rightsquigarrow \cdots\}$ maps to effective vulnerabilities represented by three-step model

Soundness of Three-step Model (b)

Yale

- More than three cache accesses
  - $\{\cdots \rightsquigarrow \star \rightsquigarrow \cdots\}$ can be divided into two parts
  - $\{\cdots \rightsquigarrow A_R \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow A_R \rightsquigarrow V_R \rightsquigarrow \cdots\}, \ldots, \{\cdots \rightsquigarrow A_1 \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \ldots, \{\cdots \rightsquigarrow V_x \rightsquigarrow V_x \rightsquigarrow \cdots\}$ can be reduced to $\{\cdots \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_R \rightsquigarrow \cdots\}, \ldots, \{\cdots \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \ldots, \{\cdots \rightsquigarrow V_x \rightsquigarrow \cdots\}$, respectively
  - $\{\cdots \rightsquigarrow (A_R/V_R \,/\, A_1 \,/\, V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R \,/\, A_1 \,/\, V_1) \rightsquigarrow \cdots\}$ maps to known vulnerabilities represented by three-step model

Exhaustive Vulnerability Search

Yale

- Explicit enumeration of all the possible three steps (6x5x5=150)
- Identify 28 types of cache attacks
  - 20 types already known or categorized
  - 8 types previously not in literature
- Can be applied to evaluate any cache architecture with CTL logic

Vulnerability Exhaustive List

Yale

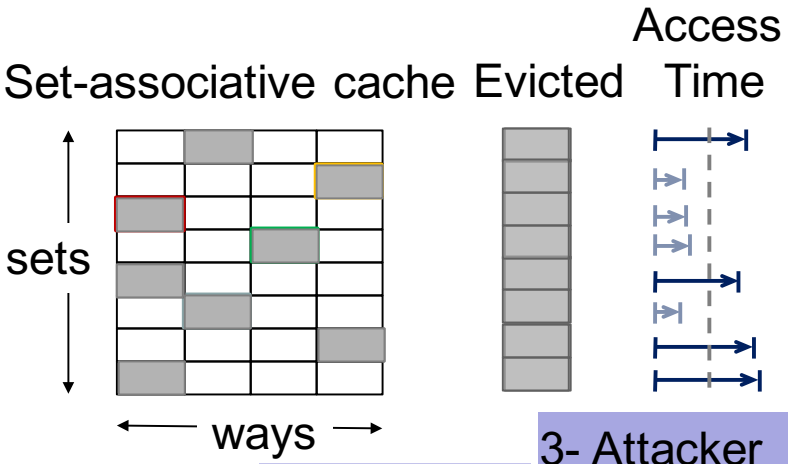
S0	S1	S2	Recognized name	Categorization
V_x	A_R	V_x		**Type A**
V_x	V_R	V_x		**Type B**
A_R	A_1	V_x		**Type C**
V_R	A_1	V_x		**Type D**
A_1	A_1	V_x		**Type E**
V_1	A_1	V_x		**Type F**
V_x	A_1	V_x	Evict+Time	Type G
A_R	V_1	V_x	Cache Collision	Type H
V_R	V_1	V_x	Cache Collision	Type I
A_1	V_1	V_x	Cache Collision	Type J
V_1	V_1	V_x	Cache Collision	Type K
V_x	V_1	V_x	Bernstein's attack	Type L
A_R	V_x	A_R	Flush+Flush	Type M
V_R	V_x	A_R	Flush+Flush	Type N

S0	S1	S2	Recognized name	Categorization
V_x	V_x	A_R	Flush+Flush	Type O
A_R	V_x	V_R	Flush+Flush	Type P
V_R	V_x	V_R	Flush+Flush	Type Q
V_x	V_x	V_R	Flush+Flush	Type R
A_R	V_x	A_1	Flush(Evict)+Reload	Type S
V_R	V_x	A_1	Flush(Evict)+Reload	Type T
A_1	V_x	A_1	Prime+Probe	Type U
V_1	V_x	A_1		**Type V**
V_x	V_x	A_1	Flush(Evict)+Reload	Type W
A_R	V_x	V_1	Cache Collision	Type X
V_R	V_x	V_1	Cache Collision	Type Y
A_1	V_x	V_1		**Type Z**
V_1	V_x	V_1	Bernstein's attack	Type AA
V_x	V_x	V_1	Cache Collision	Type AB

Vulnerability Examples

Yale

- Flush + Reload Attack (Type S, T Attack)
  - $A_R \rightsquigarrow V_x \rightsquigarrow A_1$
  - $V_R \rightsquigarrow V_x \rightsquigarrow A_1$

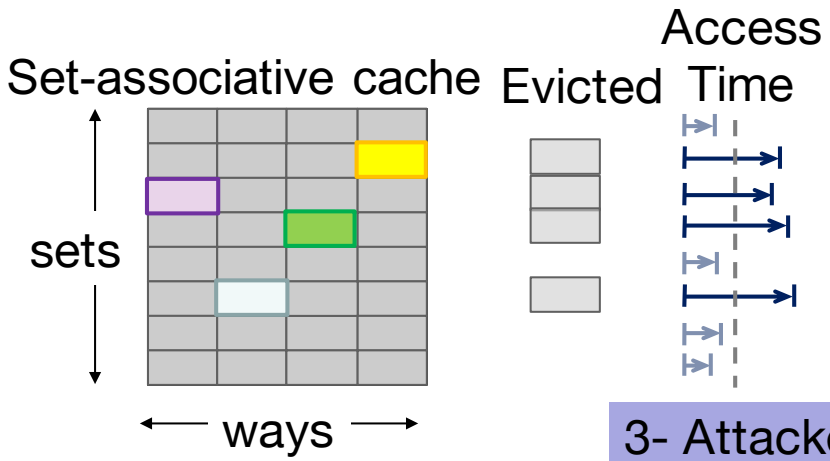


1- Flush each line in the cache

2- Victim accesses critical data

3- Attacker reloads critical data by running specific process (measure time)

- New Type V Attack
  - $V_1 \rightsquigarrow V_x \rightsquigarrow A_1$



1- Victim primes each cache set

2- Victim accesses critical data

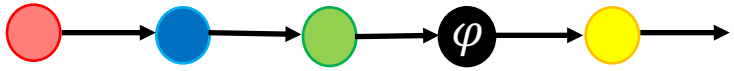
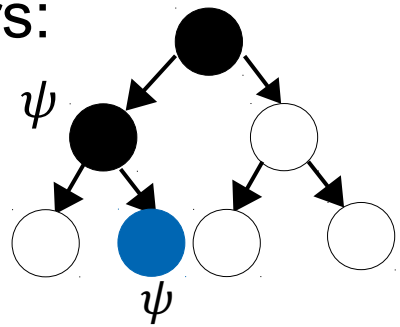
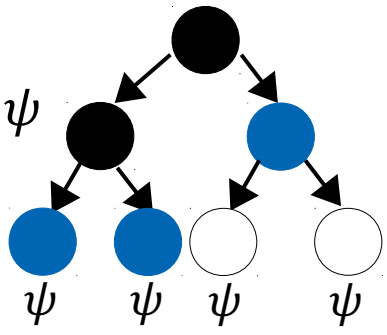
3- Attacker probes each cache set (measure time)

Computation Tree Logic

Yale

Treats time as discrete and branching

Can explore different execution paths

- Atomic propositions: ●, ●, …
- Boolean operators: $\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi$, ...
- Temporal modalities:
  - X ψ  … "next ψ"
  - φ U ψ  … "φ until ψ"
  - F φ  … "eventually φ"
  - G φ  … "always φ"
- Path quantifiers:
  - E ψ 
  - A ψ 

Three-Step Model in CTL logic

Yale

For a single cache block, model execution paths that represent vulnerabilities to attacks:

The initial state of the cache block

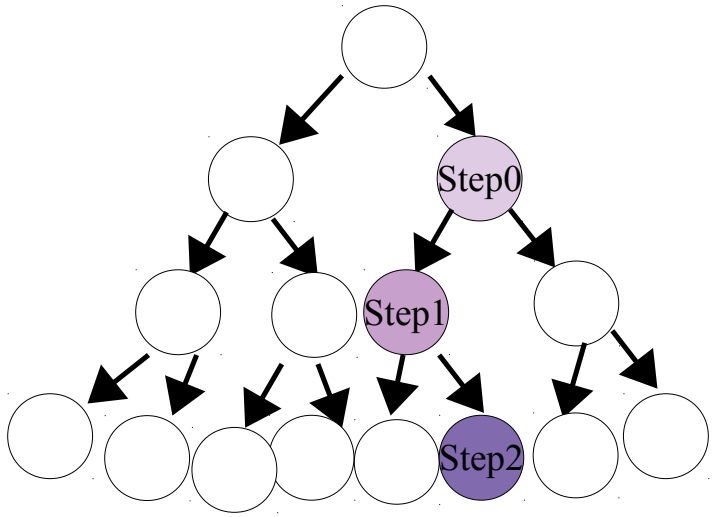
Actions of victim or attacker

Interference & final observation

$$(\mathrm{M}, \mathrm{s}) \models EF(E(E\ (Step0\ U\ Step1)\ U\ Step2))$$

Eventually there exists a path that corresponds to the vulnerability:

$Step0 \rightsquigarrow Step1 \rightsquigarrow Step2$



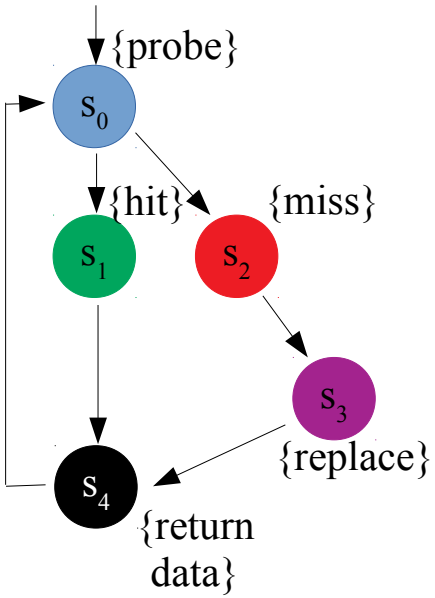
$E.g.\ A_R \rightsquigarrow V_x \rightsquigarrow A_1$

$\leftrightarrow EF(E(E\ (A_R U V_x)\ U A_1))$

Bounded Computation Tree

Yale

three-step model:



Future Work

Yale

- hardware design of secure caches
- cache state machine modeling
- checking of vulnerability in CTL logic
- improve CTL modeling

Summary

Yale

Develop Cache Access Model	Analyze Timing Vulnerabilities	Use Computation Tree Logic (CTL)
Three-step single-cache-block-access model construction	Exhaustive search for possible attacks based on three-step model	Model execution paths of the processor cache focusing on side-channel attacks

- cache state machine modeling
- checking of vulnerability in CTL logic
- improve CTL modeling

Thank you!

back up slides

Soundness of Three-step Model (a)

Yale

- One cache access
  - Interference does not exist
- Two cache accesses
  - Same as three-step model with $Step0$ to be " $\star$ "
  - None of them can form an attack
- Three cache accesses
  - Exhaustive vulnerability Search and effective vulnerabilities derived

Soundness of Three-step Model (b)

Yale

- More than three cache accesses
  - $\{\cdots \rightsquigarrow \star \rightsquigarrow \cdots\}$ can be divided into two parts
  - $\{\cdots \rightsquigarrow A_R \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow A_R \rightsquigarrow V_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow A_1 \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_x \rightsquigarrow V_x \rightsquigarrow \cdots\}$, ... can be reduced to $\{\cdots \rightsquigarrow A_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_R \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_1 \rightsquigarrow \cdots\}, \{\cdots \rightsquigarrow V_x \rightsquigarrow \cdots\}$, ..., respectively
  - $\{\cdots \rightsquigarrow (A_R/V_R \,/\, A_1 \,/\, V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R \,/\, A_1 \,/\, V_1) \rightsquigarrow \cdots\}$ maps to effective vulnerabilities represented by three-step model