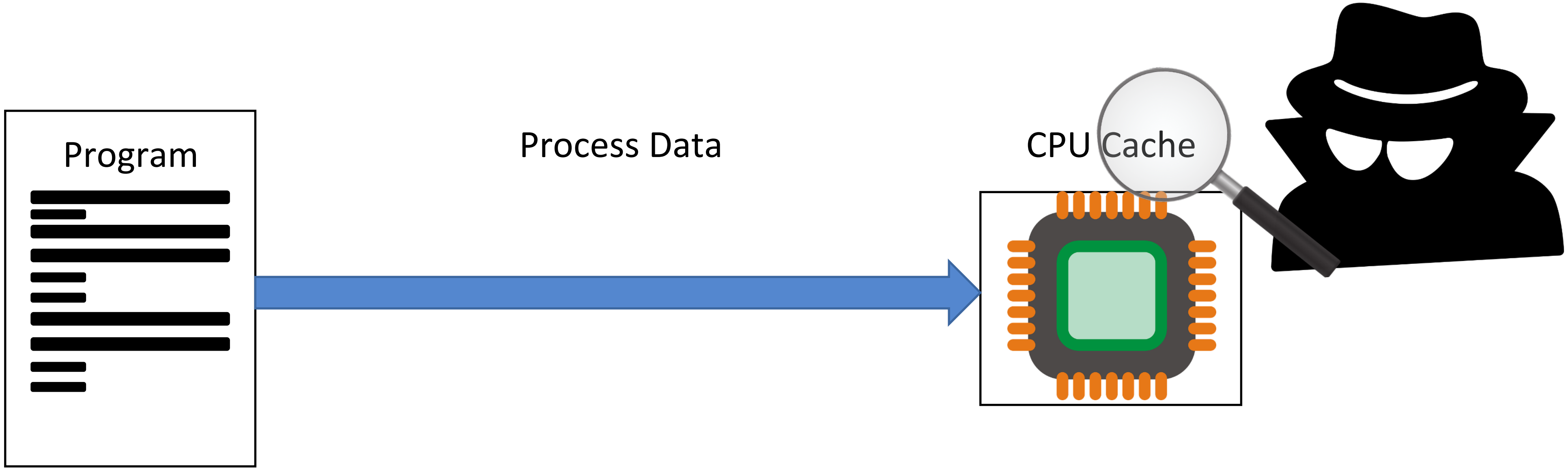
CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation

Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, Mahmut Kandemir

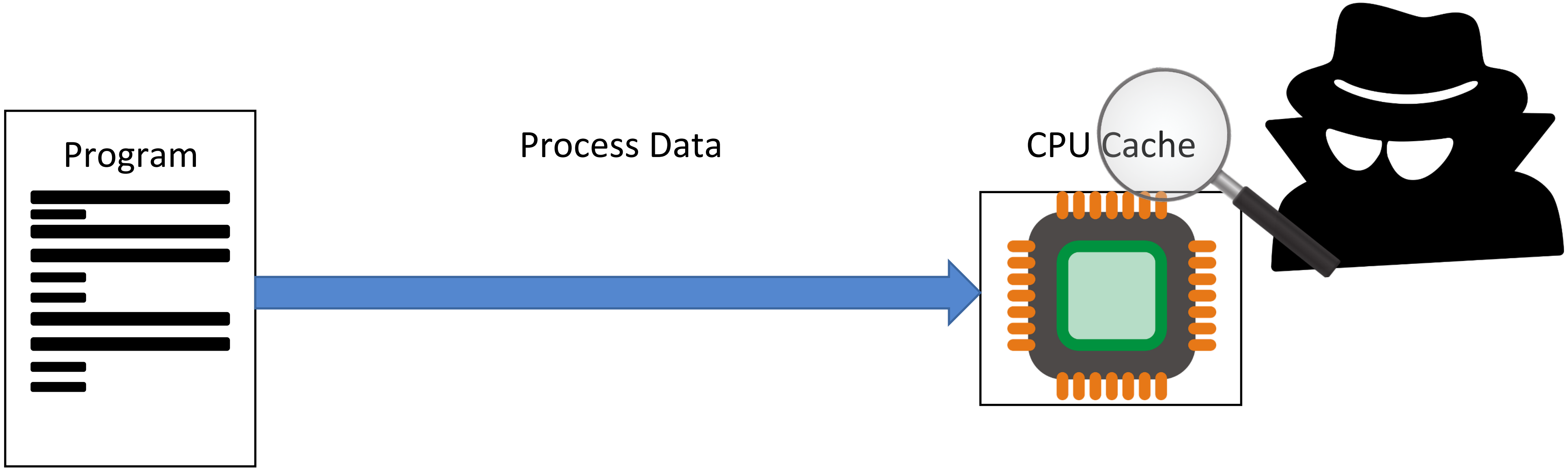
Pennsylvania State University

Cache Side Channels



- Side Channel
  - Unintentional information transfer

Cache Side Channels



- Side Channel
  - Unintentional information transfer

How Severe is the Problem?

PennState

- High band width attack
- Work on secure enclaves
- Can be launched across VM’s in the cloud

Finding vulnerabilities in code is challenging!

Security

Intel shrugs off ‘new’ side-channel attacks on branch prediction units and SGX

Researchers show how side-channel attacks can be used to steal encryption keys on A

Spectre and Meltdown: Powerful Reminders of Side Channel Attacks

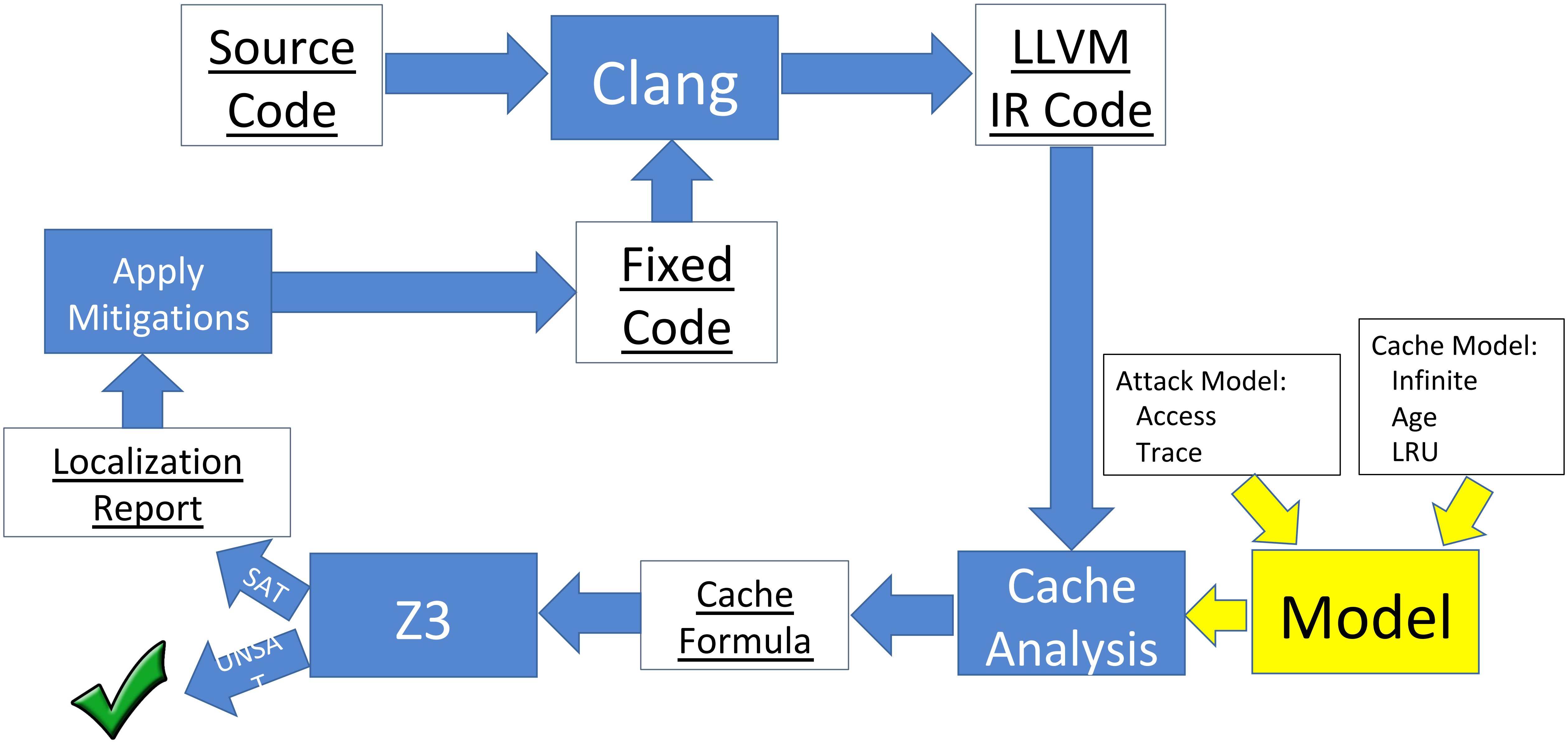
Prior Work

- CacheAudit (Doychev et al. Security ‘13)
  - Uses abstract interpretation
  - Computes upper bound on leakage
  - Does not provide location of leakage
- CacheD (Wang et al. Security ‘17)
  - Uses symbolic execution
  - Can detect where leakage happens
  - May miss side channels (not sound)
    - Requires concrete inputs
    - Does not provide fixes

Introducing CaSym

- Uses cache-aware symbolic execution
- Soundly models cache side channels
  - Memory accesses
  - Branches
- Detects cause of side channel
  - Provides simple fix mechanisms
- Flexible cache models
  - Infinite
  - Age
  - LRU

CaSym: Overview



Example: Square & Multiply

- Does modular exponentiation
- Used in asymmetric encryption
  - RSA, ElGamal, etc

Iterates over each bit of key

```
1: result = 0;
2: for(int i = expLen-1; i > 0; i--)
3: {
4:      result = result * ;
5:      result = result % mod;
6:      if((1 << i ) & exp)
7:      {
8:              result = base * result;
9:              result = result % mod;
10:     }
11: }
```

Key

Causes different observable cache states

Localization Report

Problem: Key Dependent Branch

Detected at: Line 6

Witnesses: …

Symbolic Execution

- Program variables
  - Treats all program variables symbolically
- Cache variables
  - Creates cache variable for each program variable
  - Cache variables values are determined by cache model

Toy Program

```
int a,b;
int PRIV key;

if(key == 1)
{
      a = 0;
}
else
{
      b = 0;
}
```

Cache Variables

a_{cache}
b_{cache}
key_{cache}

Verification

- Run program twice
- Cache and public variables are same between runs
- Sensitive variables must be different
- Vulnerability reported when two different cache states are achieved

Toy Program

```
int a,b;
int PRIV key;

if(key == 1)
{
        a = 0;
}
else
{
        b = 0;
}
```

Cache Variables

a_{cache}

key_{cache}

Cache Variables

a'_{cache}

key'_{cache}

Toy Program

```
int a',b';
int PRIV key';

if(key' == 1)
{
        a' = 0;
}
else
{
        b' = 0;
}
```

Cache Variables

a_{cache}
b_{cache}
key_{cache}

$\neq$

Cache Variables

a'_{cache}
b'_{cache}
key'_{cache}

Cache Models

Motivation

- Cache implementations are complex
  - Replacement policies, hierarchies, inclusivity, etc.
- Vary amongst processors

Infinite

- Treats cache as an infinite set
- Never evicts data from cache

Age

- Assigns an age to all variables
- Overapproximates real replacement policies

LRU

- Also assigns ages to all variables
- Youngest *n* variables are cached

Infinite Model Demo

Toy Program

```
int a,b;
int PRIV key;

if(key == 1)
{
        a = 0;
}
else
{
        b = 0;
}
```

Abstract Cache

key → 1

Used(key) → true

Used(a) → true

Used(b) → false

≠

Abstract Cache

key → 0

Initial

Used(key) → false

Used(a) → false

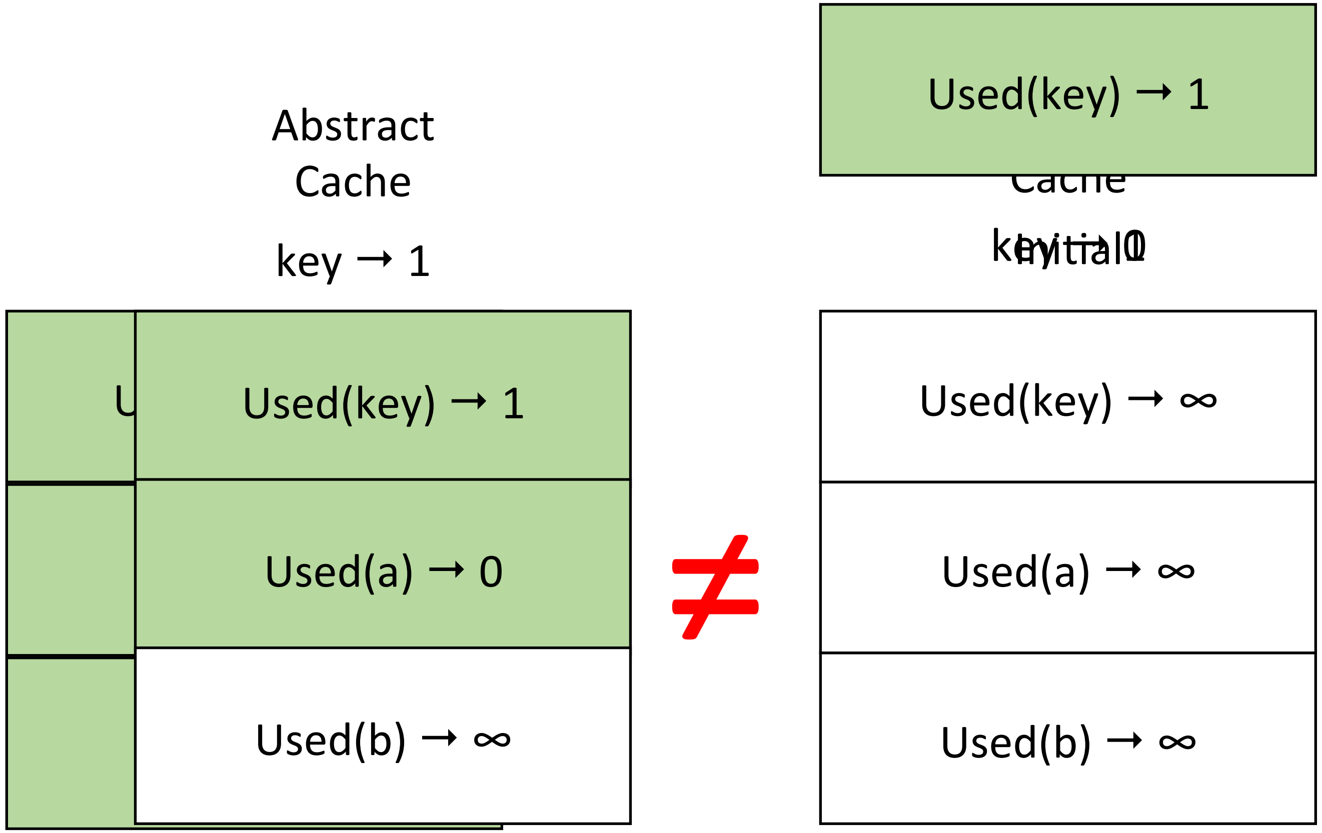
Used(b) → false

Age Model Demo

Toy Program

```
int a,b;
int PRIV key;

if(key == 1)
{
        a = 0;
}
else
{
        b = 0;
}
```



Improving Performance

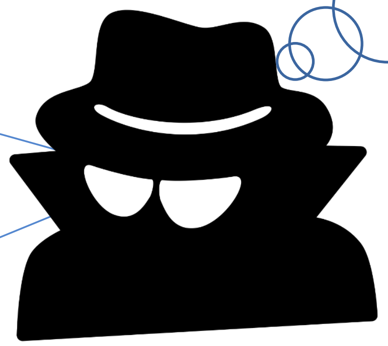
- Array reads are unconstrained
  - Uses taint analysis to check if read is sensitive
- Reset constraints
  - Breaks program into smaller chunks
  - Recomputes sensitive variables
  - Useful for loops
- Loop transformation
  - Soundly rewrite program to be loop free
  - Makes loop unrolling unnecessary

Attack Models

Access Model

Program

```
access(a);
access(b);
access(c);
access(d);
access(e);
access(f);
```



Set of Addresses:
{&a, &b, &c, &d, &e, &f}

Attack Models

Trace Model

Program

```
access(a);
access(b);
access(c);
access(d);
access(e);
access(f);
```

Sequence of Addresses:
[&a, &b, &c, &d, &e, &f]

Crypto Results: Trace

	Infinite		Age		LRU (2k)	
Benchmarks	**Found**	**Time**	**Found**	**Time**	**Found**	**Time**
AES libgcrypt	64	8.9	64	16.7	64	635
AES mbed TLS	17			17.0		
3DES libgcrypt				189		
3DES mbed TLS				73.2		
DES glibc	2		2	2.65		
UFC glibc						
Square & Multiply libgcrypt						
Square & Always Multiply libgcrypt	3		4			63
Left-to-Right Modular Exp libgcrypt	3	84.8	3	[illegible]	3	[illegible]
Totals	**269**	**217.36**	**270**	**3226.82**	**269**	**8881.85**

Order of accesses is still different

Can take significantly more time

Finds one additi
vulnerable locat

Most realistic model

Protected Results

Data cached at beginning of function

Data cached throughout function

	Preloading				Pinning			
	Infinite		Age		Infinite		Age	
Functions	**TP**	**Time (s)**	**TP**	**Time (s)**	**TP**	**Time (s)**	**TP**	**Time (s)**
AES libgcrypt	0	2.95	64	17.4	0	4.02	0	13.6
AES mbed TLS	0	1.68	17	17.4	0	2.00	0	9.60
3DES libgcrypt	0	84.0	128	170	0	0.61	0	1.53
3DES mbed TLS	0	1.53	48	65.5	0	0.03	0	1.70
DES glibc	0	0.56	2	3.15	0	0.51	0	1.79
Totals	**0**	**90.72**	**259**	**273.45**	**0**	**7.17**	**0**	**28.22**

Conclusions

- Built CaSym to automatically identify vulnerabilities in programs
- CaSym supports a variety of cache models
  - Easy to get different precision and efficiency
- Tested on an assortment of benchmarks
  - Confirm many existing vulnerabilities in crypto benchmarks
  - Verified mitigations strategies on crypto benchmarks
  - Found over 20 new potential vulnerabilities in the PostgreSQL database

Thank You!