

# Secure and Correct Systems

Todd Austin  
EECS 573 Section Introduction  
University of Michigan  
October 2010



Advanced Computer Architecture Lab  
University of Michigan

1

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

2

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



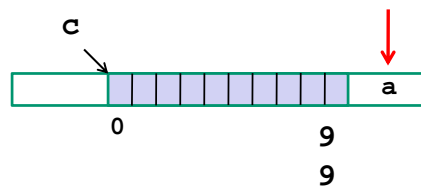
Advanced Computer Architecture Lab  
University of Michigan

3

## Memory Access Errors

- *Spatial* - Buffer overflow

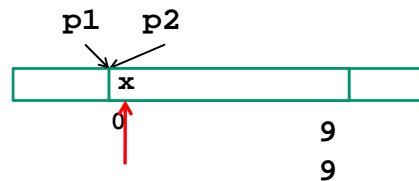
```
char *c = malloc(100);  
c[101] = 'a';
```



- *Temporal* - Dangling reference

```
char *p1 = malloc(100);  
char *p2 = p1;
```

```
free(p1);  
p2[0] = 'x';
```



Advanced Computer Architecture Lab  
University of Michigan

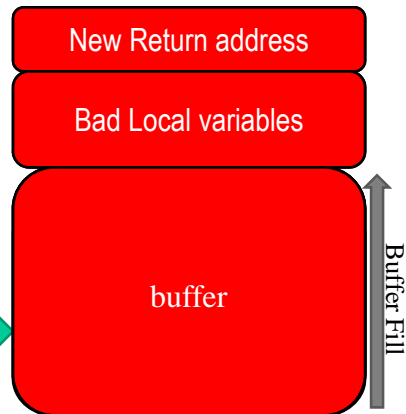
4

## Most Vulnerabilities are Due to Bugs

- Buffer overflows constitute a large class of security vulnerabilities

```
void foo()
{
  int local_variables;
  int buffer[256];
  ...
  buffer = read_input();
  ...
  return;
}
ints
```

If read\_input() reads >256



Advanced Computer Architecture Lab  
University of Michigan

5

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

6

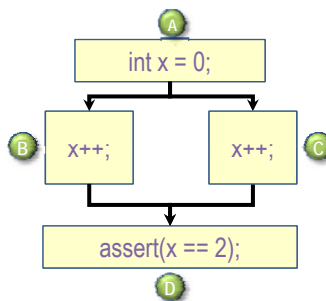
# Race Bugs

**Definition.** A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

## Example

```

A int x = 0;
B parallel_for(int i=0,i<2,++i) {
C   x++;
D }
assert(x == 2);
    
```



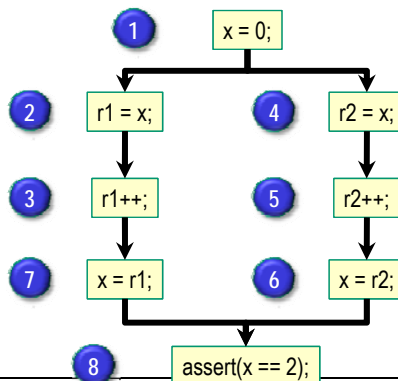
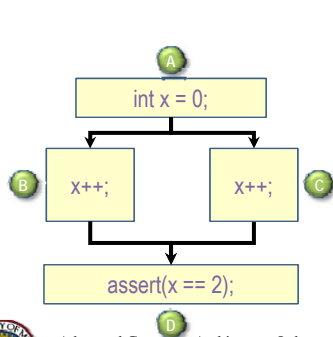
Dependency Graph



Advanced Computer Architecture Lab  
University of Michigan

# Race Bugs

**Definition.** A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.



Advanced Computer Architecture Lab  
University of Michigan

## Types of Races

Suppose that instruction **A** and instruction **B** both access a location **x**, and suppose that  $A \parallel B$  (**A** is parallel to **B**).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are *independent* if they have no determinacy races between them.



Advanced Computer Architecture Lab  
University of Michigan

9

## Today's Agenda

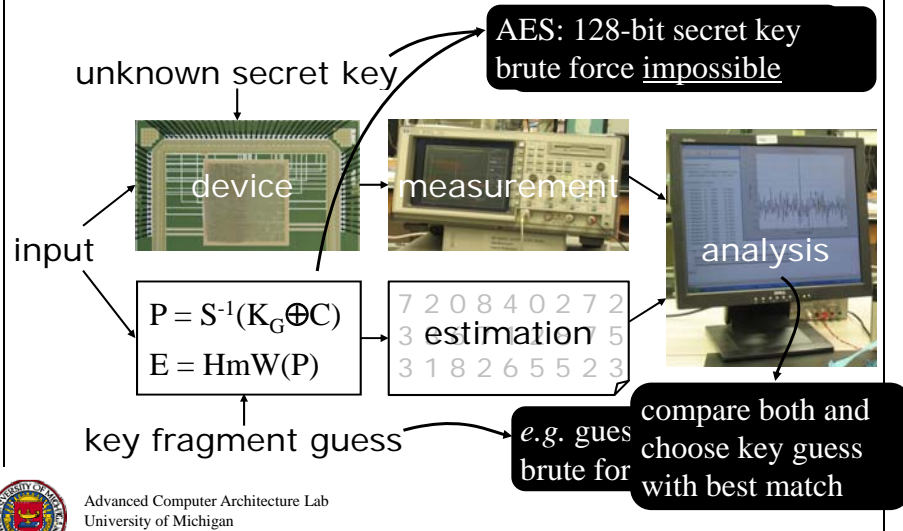
- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

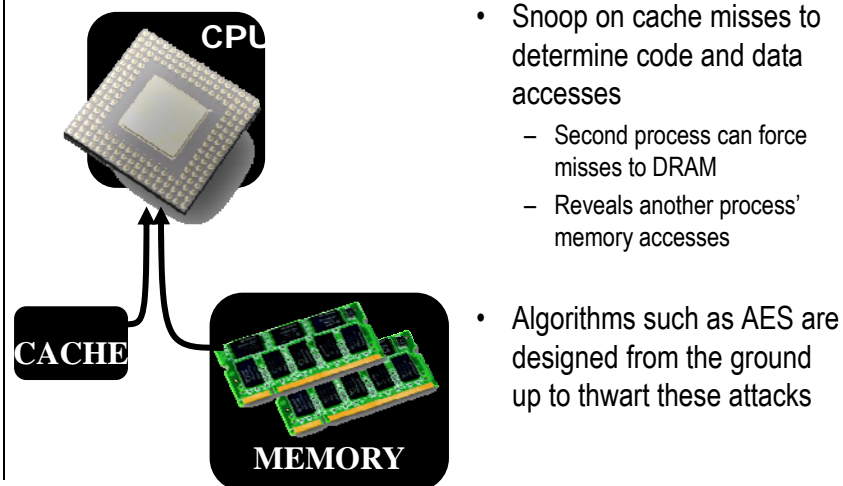
10

## Power-Based Side-Channel Attacks



11

## Cache-Based Side Channel Attacks



Advanced Computer Architecture Lab  
University of Michigan

12

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

13

## Buffer Overflow Prevention with PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: **encrypt all pointers** while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

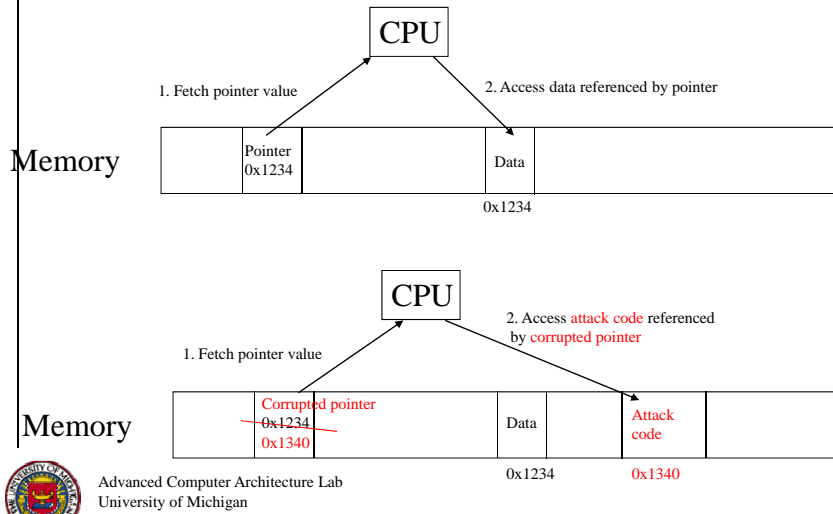


Advanced Computer Architecture Lab  
University of Michigan

14

# Normal Pointer Dereference

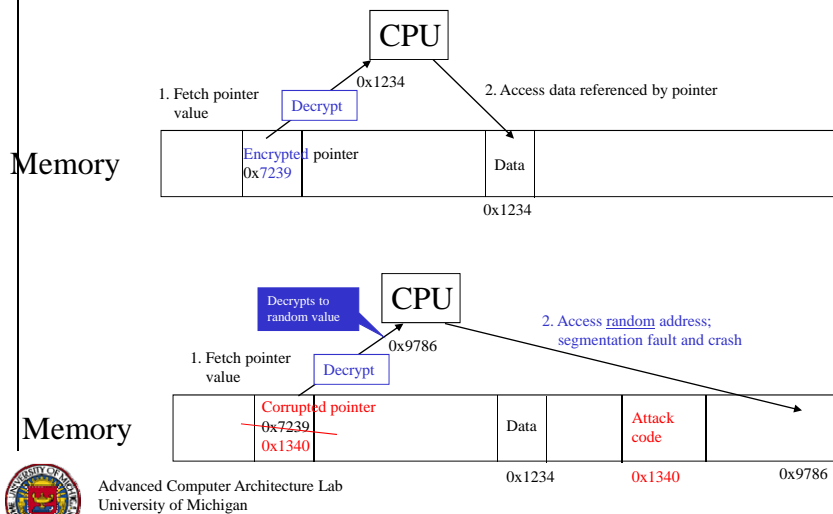
[Cowan]



Advanced Computer Architecture Lab  
University of Michigan

# PointGuard Dereference

[Cowan]



Advanced Computer Architecture Lab  
University of Michigan

## Today's Agenda

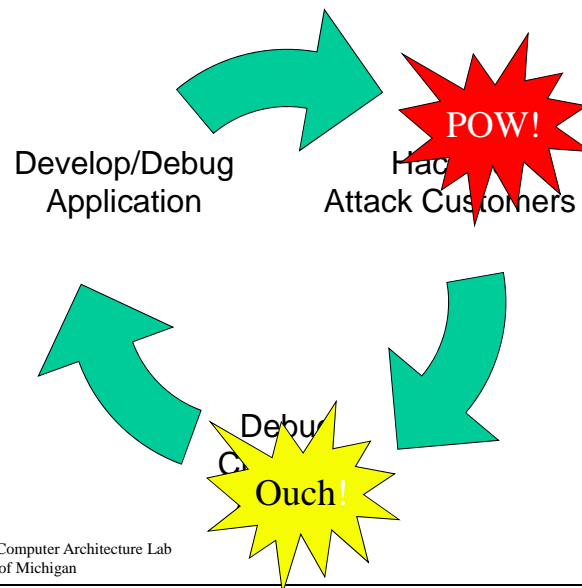
- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

17

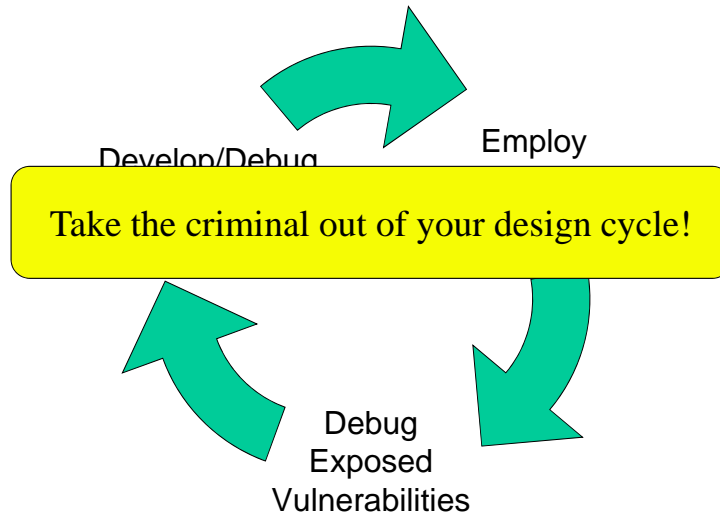
## How We Find Vulnerabilities Today



Advanced Computer Architecture Lab  
University of Michigan

18

## A Better Way - Security Vulnerability Analysis



Advanced Computer Architecture Lab  
University of Michigan

19

## Input-Related Software Faults

- Common implementation error is to improperly bound input data
  - checks are not present in many cases
  - when checks are present, they can be wrong
  - especially important for network data
- Common security exploit: buffer overflow
  - array references
  - string library functions in C
- Widespread problem:
  - 2/3 of CERT security advisories in 2003 were due to buffer overflows
  - buffer overflow bugs have recently been found in Windows and Linux

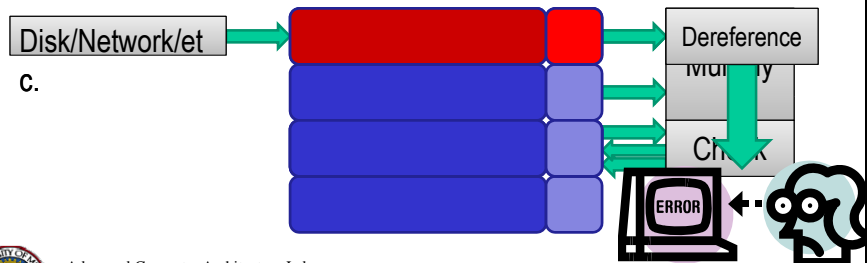


Advanced Computer Architecture Lab  
University of Michigan

20

## Finding Vulnerabilities with Taint Checking

- *Taint* data from the outside world; it's dangerous!
- Propagate taint. Bad data begets bad data.
- Clear taint by checking its validity.
- Using tainted data in a 'bad' manner = ERROR

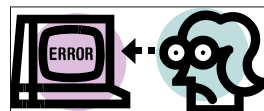


Advanced Computer Architecture Lab  
University of Michigan

21

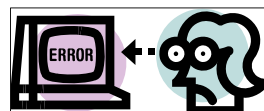
## Prevention vs. Detection

- Prevention – Stop bad things from happening



- Crash (e.g. stop data theft)
- Go into error state
- Real-time protection

- Detection – Report information about problem



- Gather data about error
- Bad usage pinpointed
- With instrumentation, can give very useful debug data



Advanced Computer Architecture Lab  
University of Michigan

22

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - **Dynamic input-bounds checking [Usenix'03]**
  - Testudo distributed dynamic debugging



Advanced Computer Architecture Lab  
University of Michigan

23

## Detecting Array Buffer Overflows

- Interval constraint variables are introduced when external inputs are read
  - Holds the lower and upper bounds for each input value
  - Initial values encompass the entire range
  - Control points narrow the bounds
  - Arithmetic operations adjust the bounds
- Potentially dangerous operations are checked:
  - Array indexing
  - Controlling a loop or memory allocation size
  - Arithmetic operations (overflow)




Advanced Computer Architecture Lab  
University of Michigan

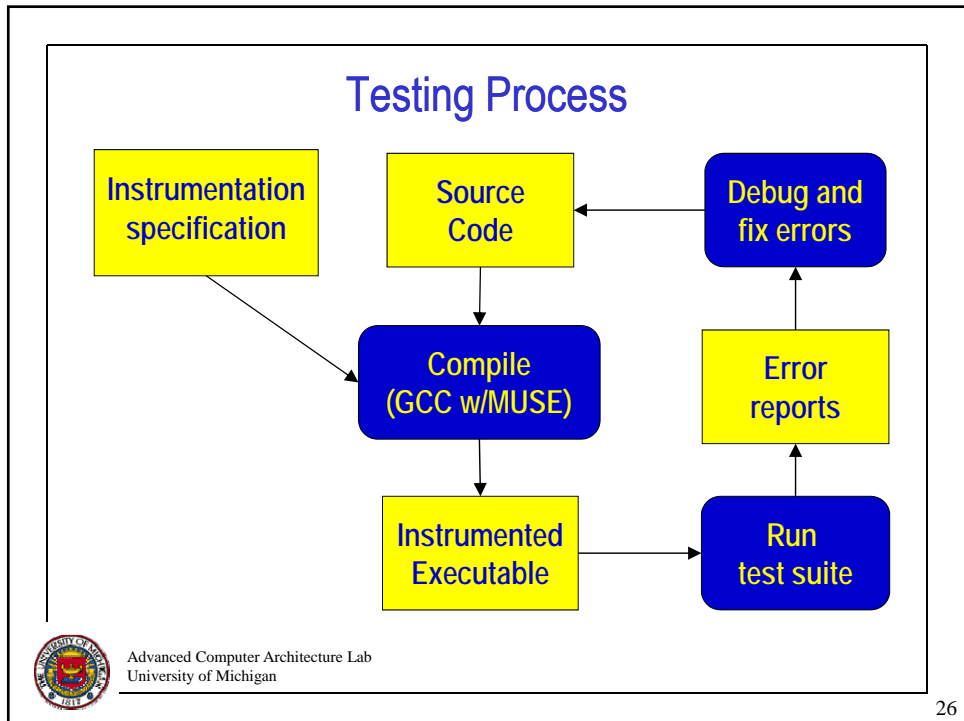
24

<u>Code Sequence:</u>	<u>Value of x:</u>	<u>Range of x:</u>
int x;		
int array[5];		
x = get_input_int();	2	$-\text{MAX\_INT} \leq x \leq +\text{MAX\_INT}$
if (x < 0    x > 4)	2	$0 \leq x \leq 4$
fatal("bounds");		
x++;	3	$1 \leq x \leq 5$
y = array[x];	3	$1 \leq x \leq 5$

ERROR! When x = 5, array reference is out of bounds!


 Advanced Computer Architecture Lab  
 University of Michigan

25



## Detection of Input-Related Software Faults

- Program instrumentation tracks data derived from input
  - possible range of integer variables
  - maximum size and termination of strings
- Dangerous operations are checked over entire range of possible values
- Found 17 bugs in 9 programs, including 2 high quality security faults in OpenSSH

Relaxes constraint that the user provides an input that exposes the bug



Advanced Computer Architecture Lab  
University of Michigan

27

## Overall Performance Results

	Base line	Unoptimized		Useless Inst. Removed		Shadow State by Name		Both	
		Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
anagram	0.06	3.15	52.50	1.32	22.00	2.24	37.33	1.12	18.67
ft									3.00
ks									5.60
yacr2									4.67
betaftpd	0.07	0.53	7.57	0.27	3.86	0.29	4.14	0.18	2.57
ghttpd	0.52	1.08	2.08	0.69	1.33	0.73	1.40	0.59	1.13
openssh	0.70	1.00	1.43	0.91	1.30	0.83	1.19	0.78	1.11
thttpd	0.15	2.57	17.13	1.78	11.87	2.14	14.27	1.82	12.13

Analysis is effective, but EXPENSIVE!



Advanced Computer Architecture Lab  
University of Michigan

28

## Today's Agenda

- Bugs and Security Vulnerabilities
  - Memory access errors
  - Race bugs
  - Side channel attacks
- Exploit prevention techniques
  - Buffer overflow protection
- Security vulnerability analysis
  - Taint tracking
- Examples from My Work
  - Dynamic input-bounds checking
  - **Testudo distributed dynamic debugging [MICRO'08]**

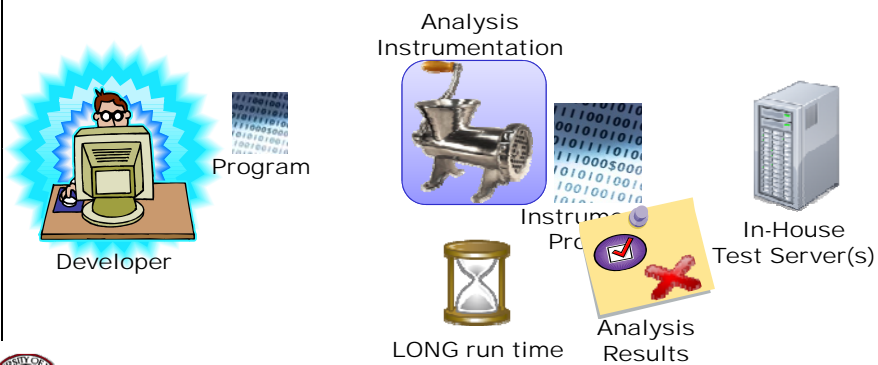


Advanced Computer Architecture Lab  
University of Michigan

29

## Software Dynamic Analysis for Security

- Valgrind, Rational Purify, DynInst
  - + Multiple types of tests, runtime protection
  - Extremely high runtime overheads

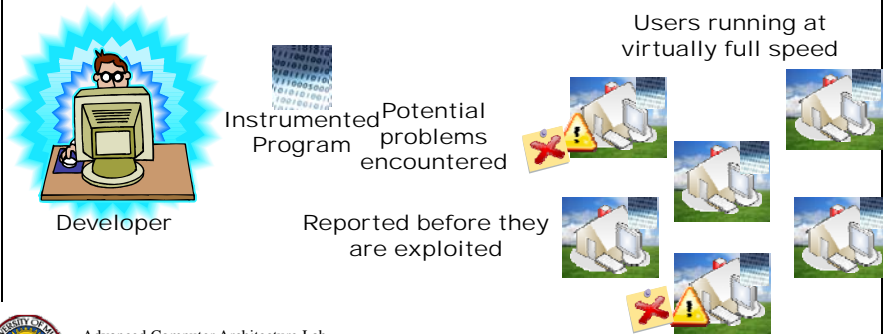


Advanced Computer Architecture Lab  
University of Michigan

30

## Testudo: Dynamic Distributed Debug [MICRO'08]

- Split analysis across population of users
  - + Low HW cost, low runtime overhead, runtime information from the field
  - Analysis only

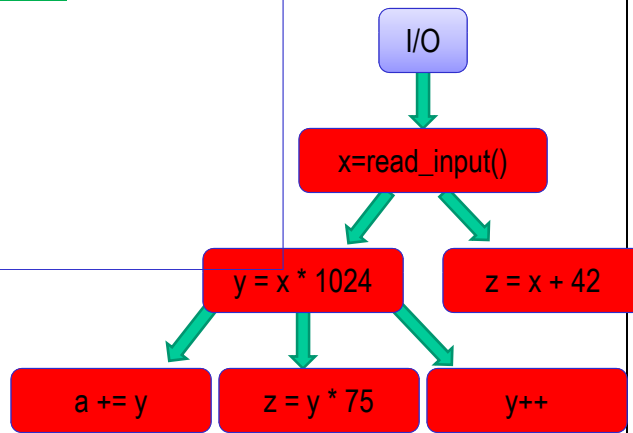


Advanced Computer Architecture Lab  
University of Michigan

31

## Traditional Brute-Force Vulnerability Analysis

```
void function(int &y, int &z, int &a)
{
    int x = read_input();
    y = x*1024;
    a += y;
    z = y * 75;
    ...
    y++;
    z = x+42;
    return;
}
```



**Key:**  
■ = has shadow value



Advanced Computer Architecture Lab  
University of Michigan

32

## Sampled Dataflow Analysis

HW limitations force us to ignore some shadow value stores

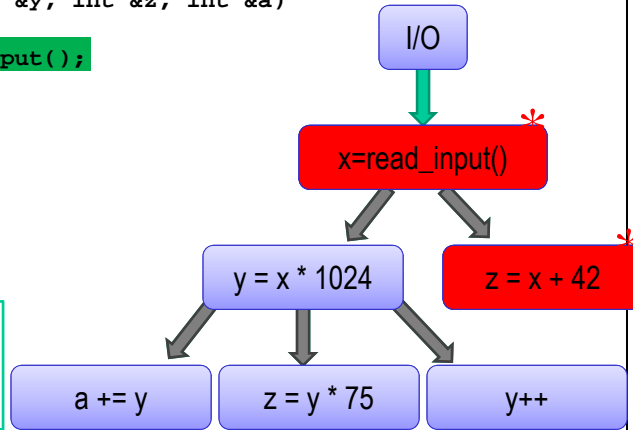
1<sup>st</sup> execution:

```
void function(int &y, int &z, int &a)
{
  int x = read_input();
  y = x*1024;
  a += y;
  z = y * 75;
  ...
  y++;
  z = x+42;
  return;
}
```

**Key:**  
■ = has shadow value  
■ = no shadow value  
\* = globally observed shadow value



Advanced Computer Architecture Lab  
University of Michigan



33

## Sampled Dataflow Analysis

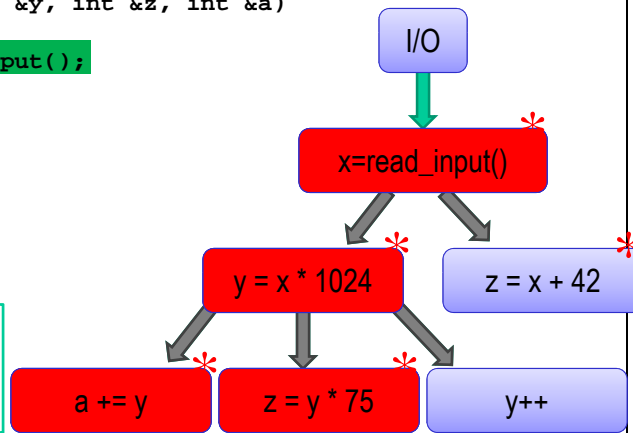
2<sup>nd</sup> execution:

```
void function(int &y, int &z, int &a)
{
  int x = read_input();
  y = x*1024;
  a += y;
  z = y * 75;
  ...
  y++;
  z = x+42;
  return;
}
```

**Key:**  
■ = has shadow value  
■ = no shadow value  
\* = globally observed shadow value



Advanced Computer Architecture Lab  
University of Michigan



34

## Sampled Dataflow Analysis

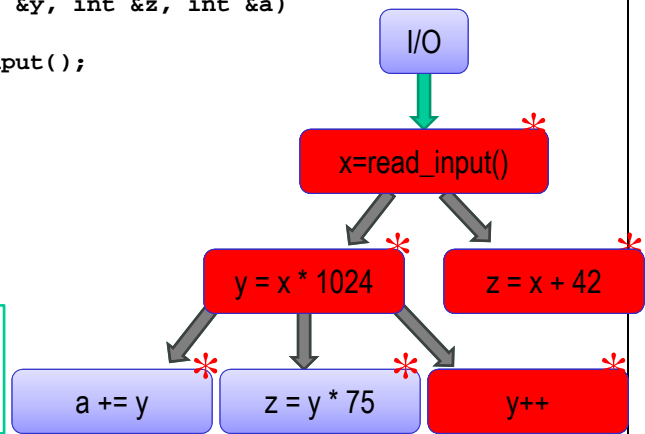
3<sup>rd</sup> execution:

```
void function(int &y, int &z, int &a)
{
  int x = read_input();
  y = x*1024;
  a += y;
  z = y * 75;
  ...
  y++;
  z = x+42;
  return;
}
```

**Key:**  
■ = has shadow value  
■ = no shadow value  
\* = globally observed shadow value

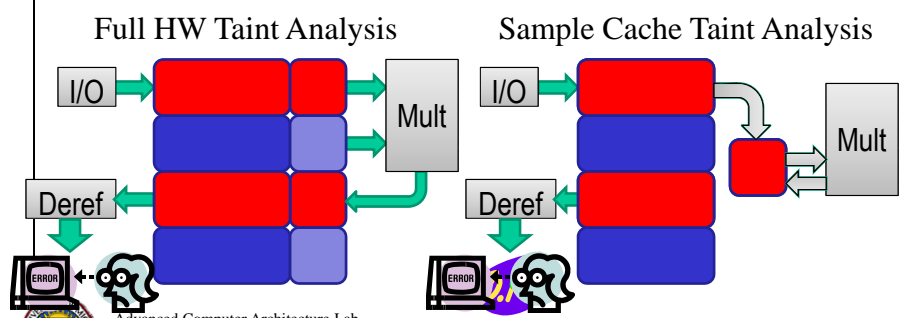


Advanced Computer Architecture Lab  
 University of Michigan



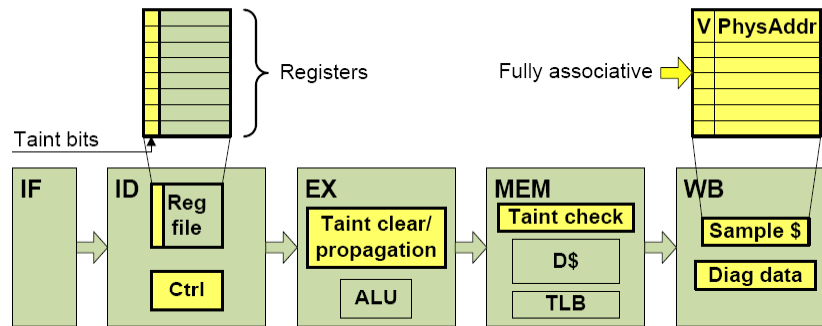
## Fix the cost! Using the Sample Cache

- Hardware, but follow only a few randomly selected taint bits.
- Constant, small HW overhead
- Requires many runs of program to find bugs



Advanced Computer Architecture Lab  
 University of Michigan

## Small Pipeline Additions



Advanced Computer Architecture Lab  
University of Michigan

37

## The ups and downs of sampled analysis

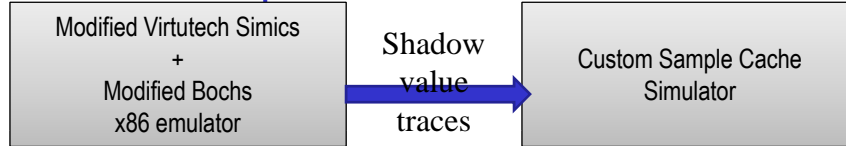
- + Constant small hardware+memory overhead
- + Low runtime overhead
- + Low overheads → Easy deployment
- + Highly-used software becomes highly tested
- Can require many runs (probabilistic)
- Used for detection, **not** prevention.



Advanced Computer Architecture Lab  
University of Michigan

38

## Experimental Framework



- Insecure programs (with exploits), including:
  - TIFF image engine
  - Eggdrop IRC bot
  - Lynx web browser
  - PDF library
  - Simulated SQL injection
- CACTI v5.0 used for cache estimation

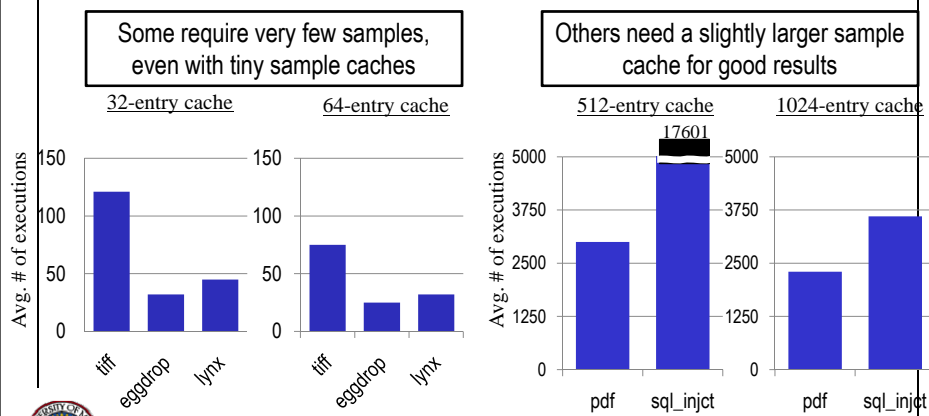


Advanced Computer Architecture Lab  
University of Michigan

39

## How Many Runs Will I Need?

How many times must a program be sampled before completely seeing all of its dataflows with high statistical confidence?

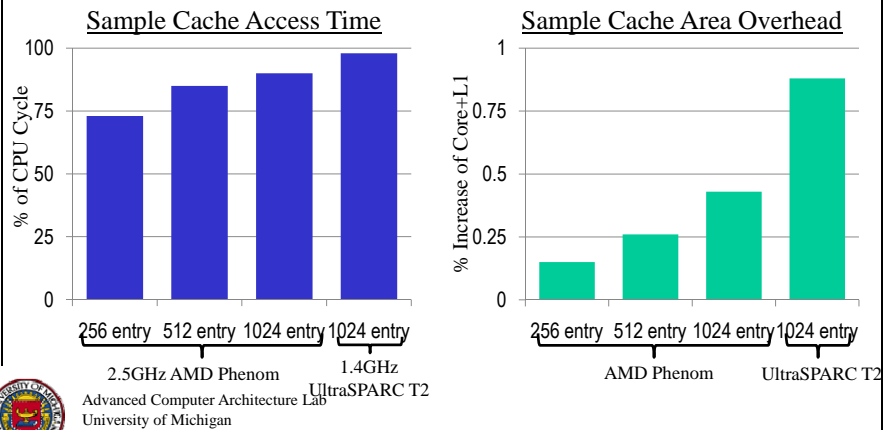


Advanced Computer Architecture Lab  
University of Michigan

40

## How Much Will It Cost?

- No overhead outside of the CPU core
- No perceivable change in clock period of modern CPU



Advanced Computer Architecture Lab  
University of Michigan

41

## Insights from Testudo

- Spinning taint analysis into a new domain created potential & number of challenges.
- This cache system brings up new questions
- Analytic model presented interesting challenges: can it get better?
- HW for SW checking is an interesting paradigm



Advanced Computer Architecture Lab  
University of Michigan

42