# Secure Systems 2.0:

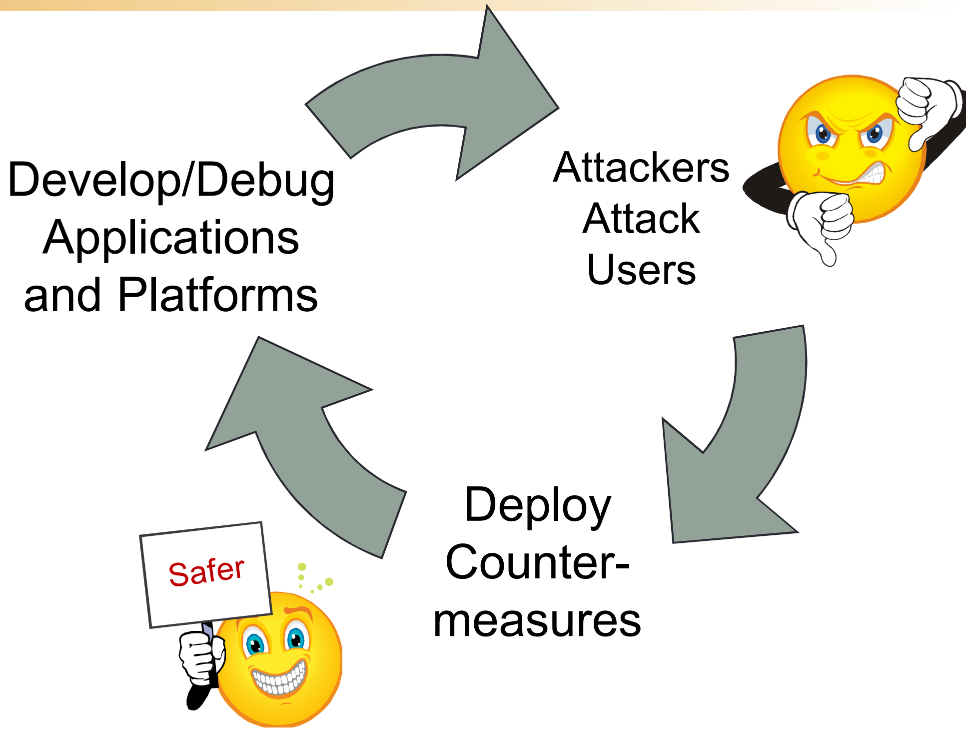
## Revisiting and Rethinking the Challenges of Secure System Design

Todd Austin
University of Michigan

# The Security Arms Race

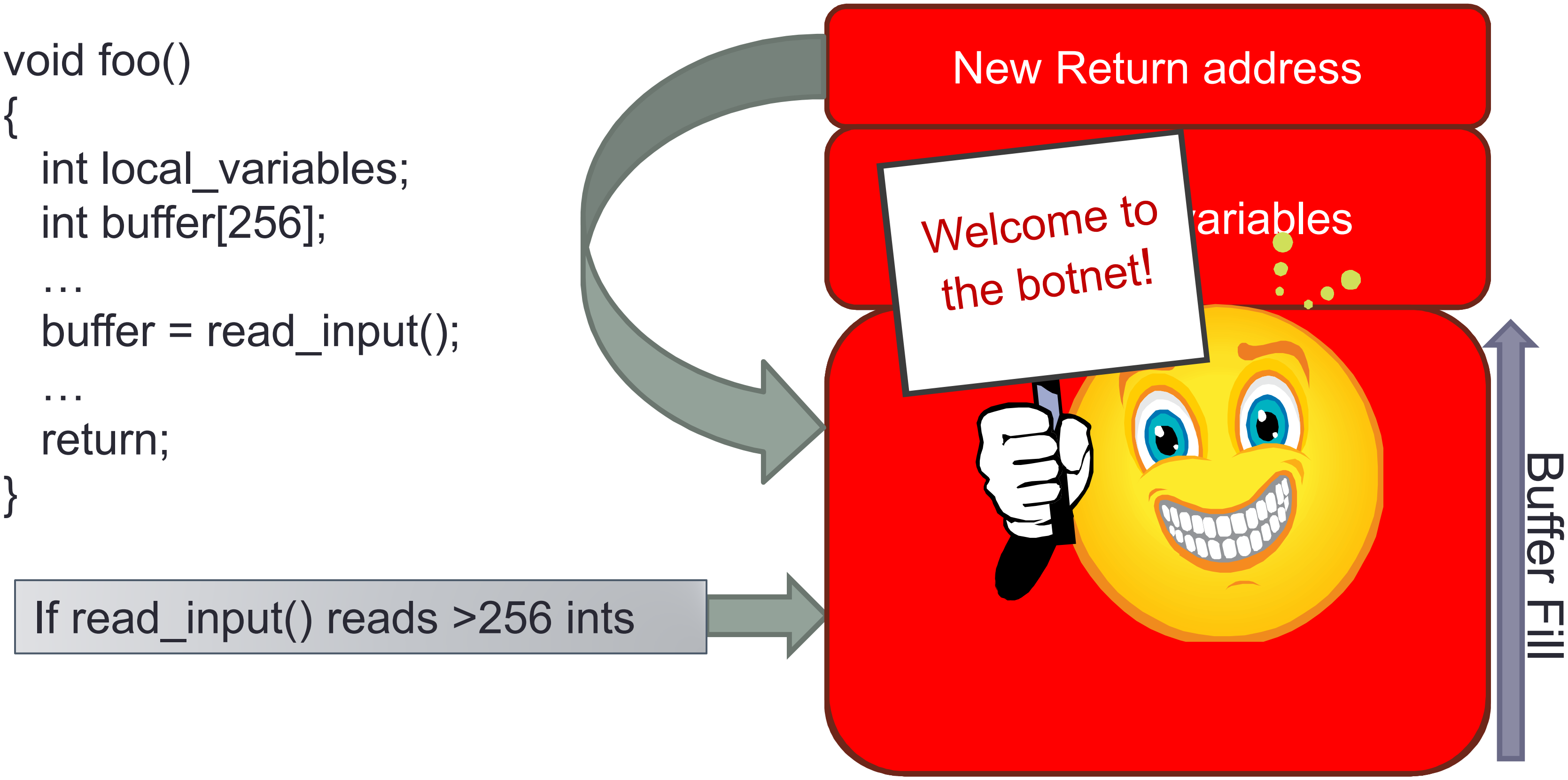
- Question: Why are systems never safe?
  - We deploy our designs
  - Attackers attack
  - We deploy countermeasures
  - Rinse and repeat

- Let's see an example of the arms race for ***code injection***



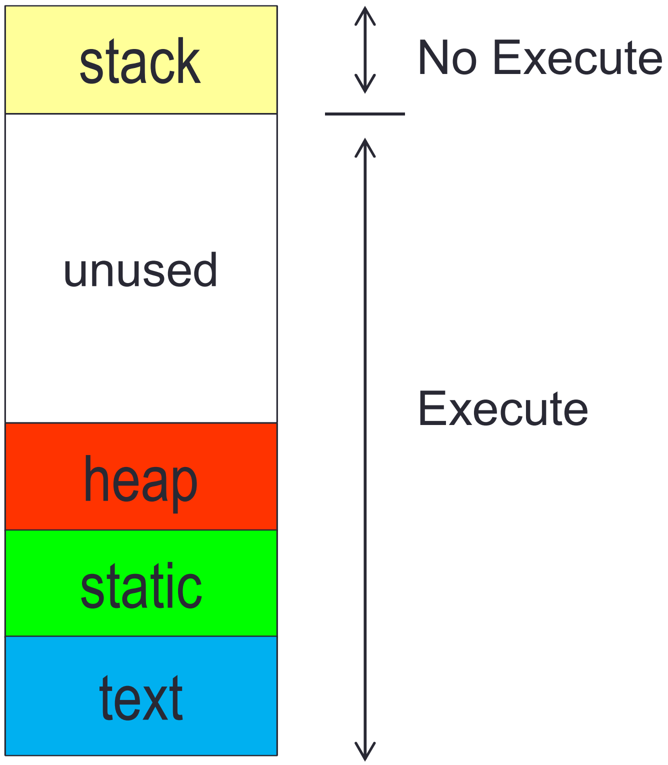
# In the Beginning: Buffer Overflow Attacks

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



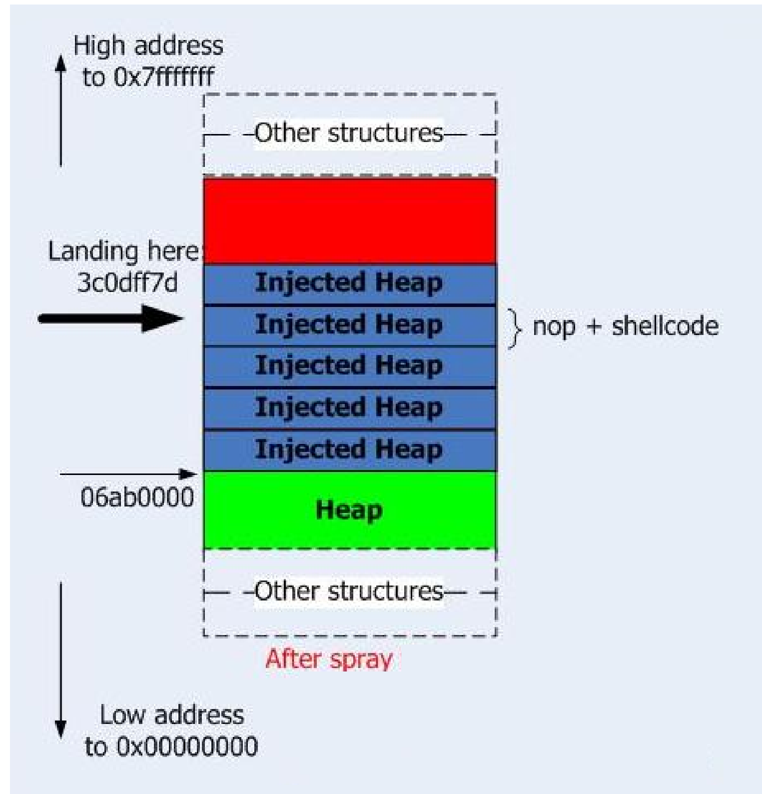
# Countermeasure: No-Execute (NX) Stacks

- Eliminates stack code injection by stopping code execution on stack
- Can be a problem for some safe programs, e.g., JITs
- Often, a general mechanism via e(x)ecute bit in page table PTEs



# Enter: Heap-spray Attacks

- Inject executable data into heap, then perform random stack smash
  - Example: generate many strings in Javascript that are also attack code
- Generously large heap sprays are easily found

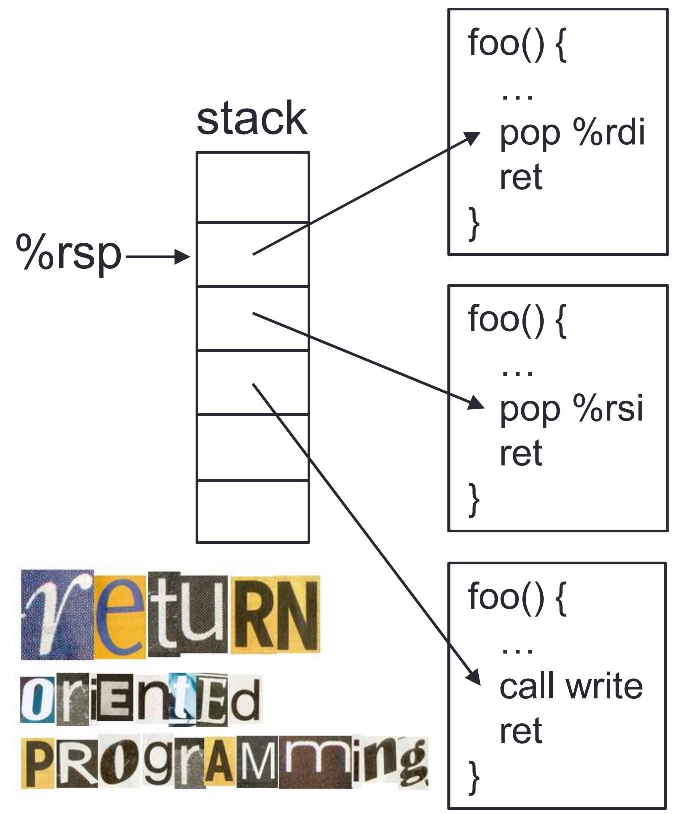


# Countermeasure:
# Address Space Layout Randomization (ASLR)

- At load time, insert random-sized padding before all data, heap, and stack sections of the program
- Successfully implementing a heap-spray requires guessing the heap location ***on the first try***
- Provides more safety on 64-bit systems
- Often, ***code placement is not randomized*** due to position-independence requirement

# Enter: Return-Oriented-Programming Attacks

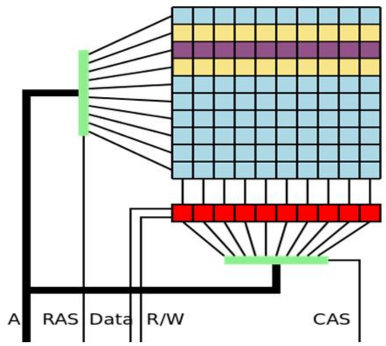
- Smash stack with many returns to the tails of functions
- Returns ***stitch together a new program*** (from existing code) using the tails of functions
- This form of code injection doesn't inject new code, but reuses the code that is already there!



# Hardware is Catching Up Fast

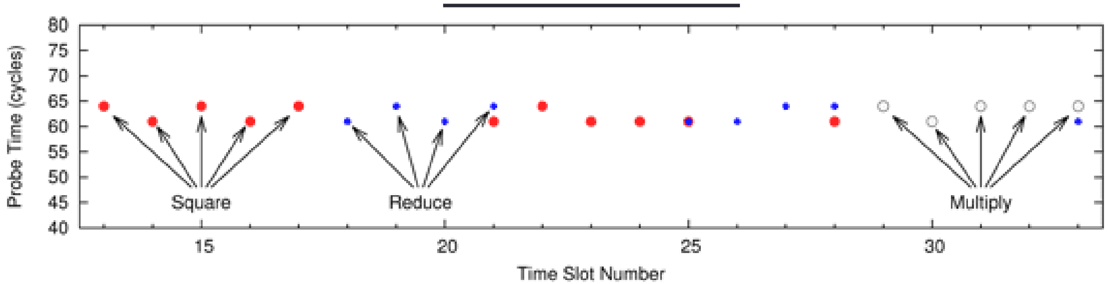
- Growing list of hardware vulnerabilities calls into question the extent to which hardware can establish a ***root of trust***

Rowhammer

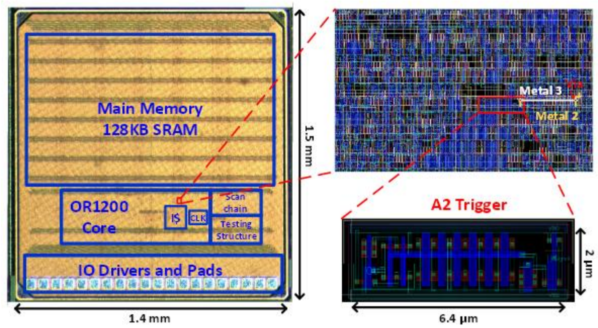



Cold Boot (DDR3/4)

Flush+Reload

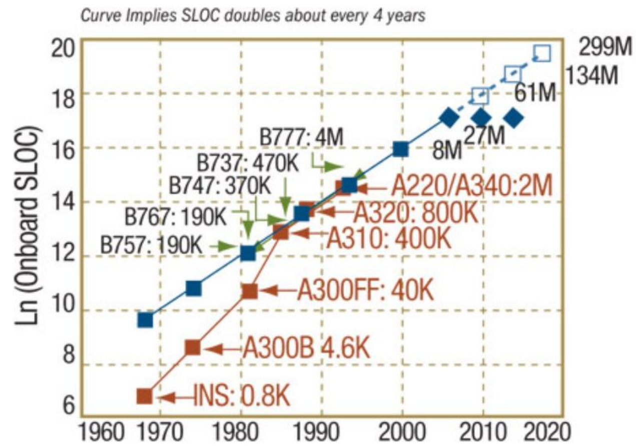



A2 Malicious Hardware



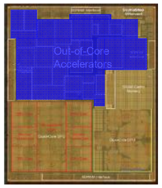
# Why Does the Race Never End?

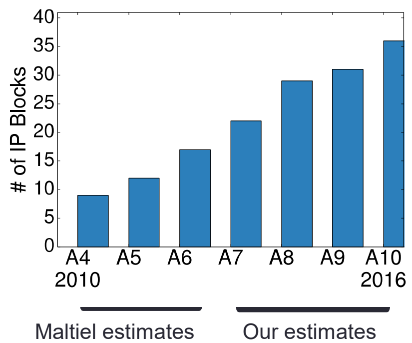
- Traditional ***additive methods*** add protections to thwart attacks
  - Stack smash begat NX bit begat heap spray begat ASLR begat ROP…
  - Verifying an additive measure requires a ***nonexistence proof***
    vulnerabilities
  - *For all <programs, inputs>, there exists no unchecked vulnerability*




- ***Principled designs*** are not generally possible due to immense ***attack surface***
  - Created by software and hardware complexity
  - Increasing complexity worsens challenge

A10





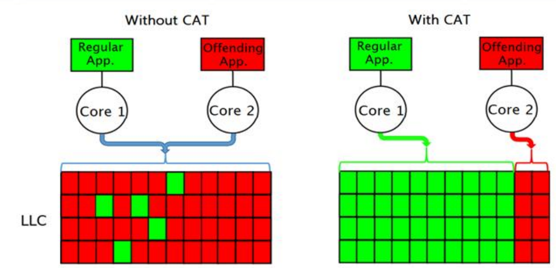
# When Good Protections Go Bad: CAT-Assisted Rowhammer

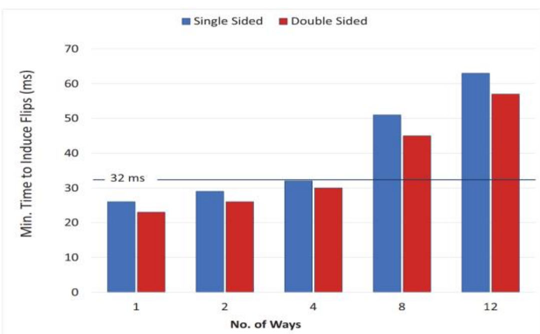
- Current rowhammer protections are effective
  - When used *in tandem*

- CAT technology was made (in part) to prevent VM denial-of-service
  - Works well in this regard
  - Also works well to speed up rowhammer!

- Rowhammer attack approach:
  1. Pose as a VM “noisy neighbor” and get LLC cache access restricted by CAT
  2. Rowhammer using single-ended CFLUSH-free attack mode
- Defenses?
  - Most recent defenses work: ANVIL, PARA

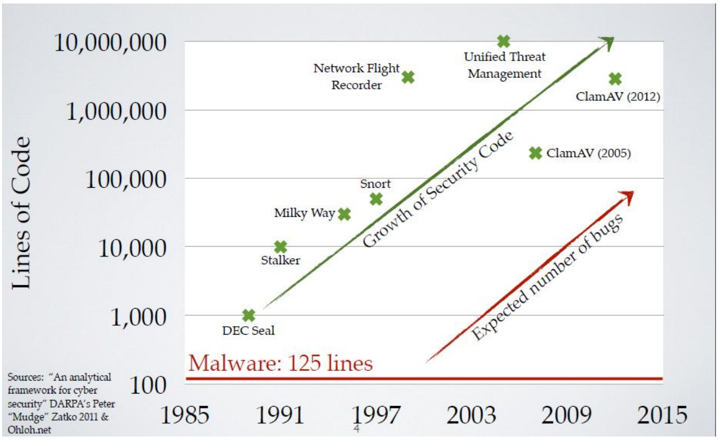
| | CLFLUSH-based | | CLFLUSH-Free ( Aweke et al, ASPLOS 2016, Gruss et al, DIMVA 2016) |
|---|---|---|---|
| | **Single-sided** (kim et al, ISCA 2014) | **Double-sided** ( Seaborn et al, Blackhat 2015) | |
| Double Refresh Rate | × | ✓ | × |
| Restricted Pagemap | ✓ | × | × |
| Disabled CLFLUSH | × | × | ✓ |

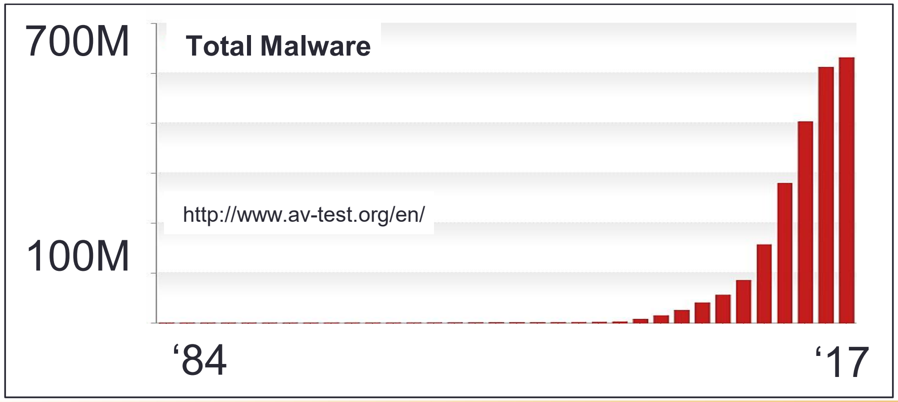


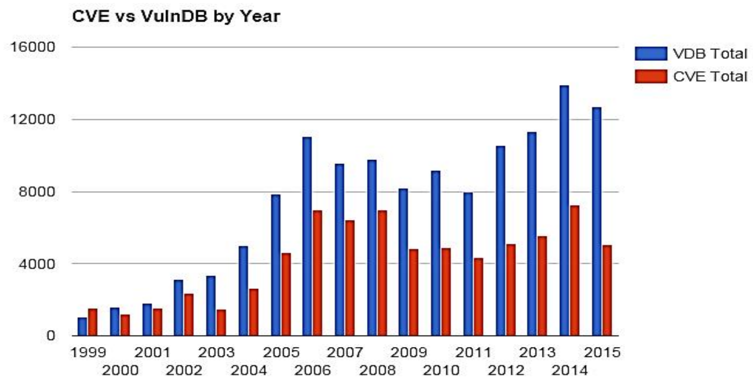



# Attackers Have the Upper Hand

- Attacking is ***fundamentally easier*** than protecting against attacks
  - Attacking requires ***one*** bug/vulnerability
  - Protecting requires ***100% coverage*** of all bugs/vulnerabilities (mostly incomplete)
- Consequently, attacks and vulnerabilities are on the rise



# My Goal Today is to Suggest a Better Way

- ***Subtractive security*** techniques ***remove functionality*** from the system necessary to implement classes of attacks

- The approach is a ***principled approach*** to achieving complete coverage of all vulnerabilities for non-trivial systems

- Demonstrated via a ***single-instance constructive proof***

# Subtractive Security Techniques

- ***Additive methods*** add protections to thwart attacks
  - Verifying additive measures requires a ***nonexistence proof***
  - *For all <programs, inputs, vulnerabilities>, there exists no unchecked vulnerability*
- ***Subtractive methods*** remove “functionality” needed to implement a class of attacks
  - Rebuild the ***subtractive design*** to work without functionality
  - Implementation is an ***constructive proof*** that approach works
  - ***Optimize subtractive design*** to negate overheads
  - Resulting system is ***immune to targeted class of attacks***
- Why does this work so well?
  - ***Attack functionality differs radically*** from normal activity
  - Constructive proofs are naturally ***scalable and approachable proof*** techniques

# Two Examples...

- Control-data isolation (CDI), to stop code injection
- Ozone zero-leakage execution mode, to stop timing side channels

# Example #1: Control-Data Isolation

[CGO'15]

- Code injection ***requires*** indirection
- All ***indirection removed***, uses ***whitelisted*** direct jumps to ***thwart all code injection***
  - ***Direct***, as specified by programmer
  - ***Validated***, via whitelisting
  - ***Complete***, no indirection remains
- System supports run-time code gen and dynamic libraries


**Vulnerable Code**

```
Int foo() {
  /* fptr */
  fptr = %cx;
  call *fptr;
Work:
  ...        }
```

```
Int bar() {
  return; }
```

```
Int baz() {
  return; }
```

**Control-Data Isolated Code**

```
Int foo() {
  /* fptr */
  fptr = %cx;
  if(*fptr==bar)
   call bar;
Ret_1:
  else if(*fptr==baz)
   call baz;
Ret_2:
  else
   call InvalidCFG!
Work:
  ...        }
```

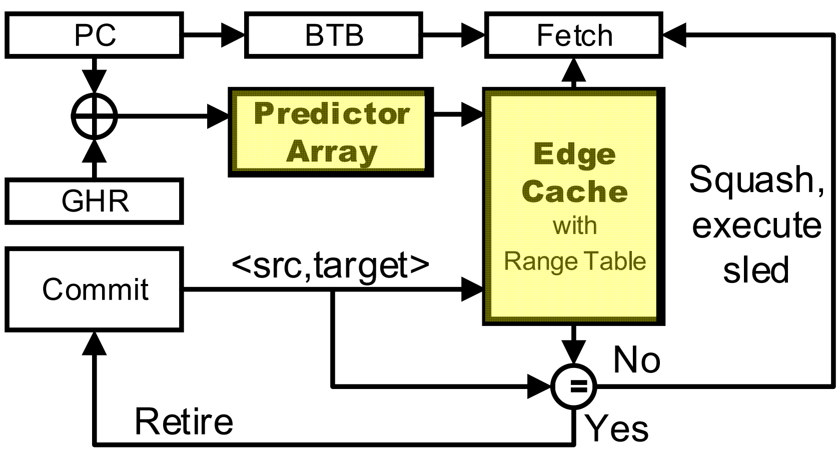
```
Int bar() {
 if([%sp] == Ret_1)
  inc %sp;
  jump Ret_1;
 else
  call InvalidCFG!;}
```

```
Int baz() {
 if([%sp] == Ret_2)
  inc %sp;
  jump Ret_2;
 else
  call InvalidCFG!;}
```

# Hardware Support Erases Overheads

[MICRO'15]

- Software-only approach experiences 7% slowdown
  - Due to indirect whitelist validation that occurs at all indirect jumps
- Edge cache memoizes edge validations, doubles as predictor
  - With range table, 6kB ***edge cache reduces slowdowns to 0.3%***
  - Indirect target prediction ***cuts misprediction rate in half*** over simple BTB



# Example #2:
# Ozone Zero-Timing-Leakage Architecture

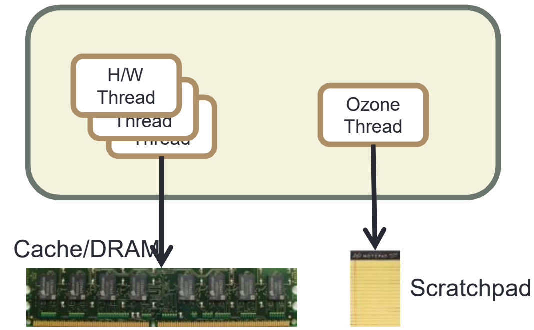
- Even carefully designed systems leak info about internal computation
  - Example: safes can be cracked by carefully listening to the tumblers
- Clever attackers can utilize leaked information to gain secrets
  - If not directly, use statistical methods
- Current ***protections are additive***
  - ***Add delays*** to the system to hide timing
  - ***Add superfluous activities*** to hide actions
  - Side channels persist despite measures

# Ozone Zero-Timing-Leakage Architecture

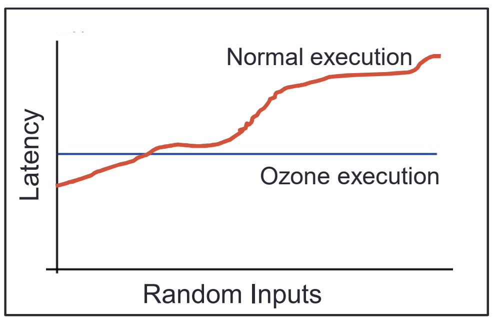
[HOST’17]

- ***Functionality removed:*** all characteristics that create timing channels
  - *Common case **not** optimized*
  - ***No** resource sharing*
  - ***No** fine-grained timing analysis*
- Implementation approach:
  - Ozone H/W thread runs in fixed time
  - No complex (hammock) control, use static predictor
  - Only access to scratchpad memory
  - Does not share resources
  - Not subject to context switches
- ***Zero timing leakage*** and ***10x faster*** than additive approaches

Ozone Architecture




Execution Characteristics



# Challenges and Opportunities

- To what extent can subtractive security stop vulnerabilities?
  - Demonstrated for code injection and timing leakage
  - Could it work for rowhammer, memory side channels, and malicious hardware?
- To what extent will these techniques be composable?
  - prot(code injection) + prot(timing leakage) ?= no code inject, no leakage
- To what extent will these techniques be deployable?
  - Code-data isolation requires complete overhaul of build tool chain
  - Ozone zero-leakage architecture somewhat restricts code expression
  - Will system designers pay for these technologies?

# Conclusions

- My challenge to you: let's get the upper hand back from attackers
  - Requires a principled approach that ***shuts down*** vulnerabilities
  - This is simply ***intractable for additive security measures***

- ***Subtractive security*** measures are a ***principled approach*** that are simpler to validate
  - Creation of a working system constitutes a constructive proof
  - Has already been demonstrated for multiple vulnerabilities

- Would this approach address your critical vulnerabilities?

# Looking Ahead

- Exploring new models of “principled design”


Our new tools:

- Lies and deception
- Misdirection and bewilderment
- False hope and broken dreams

Challenges:

- Scalability and complexity
- Composability
- Soundness and completeness

# Questions?