Securing Software Systems by Preventing Information Leaks

Kangjie Lu

Georgia Institute of Technology

Computer devices are everywhere

Foundational software systems

iOS

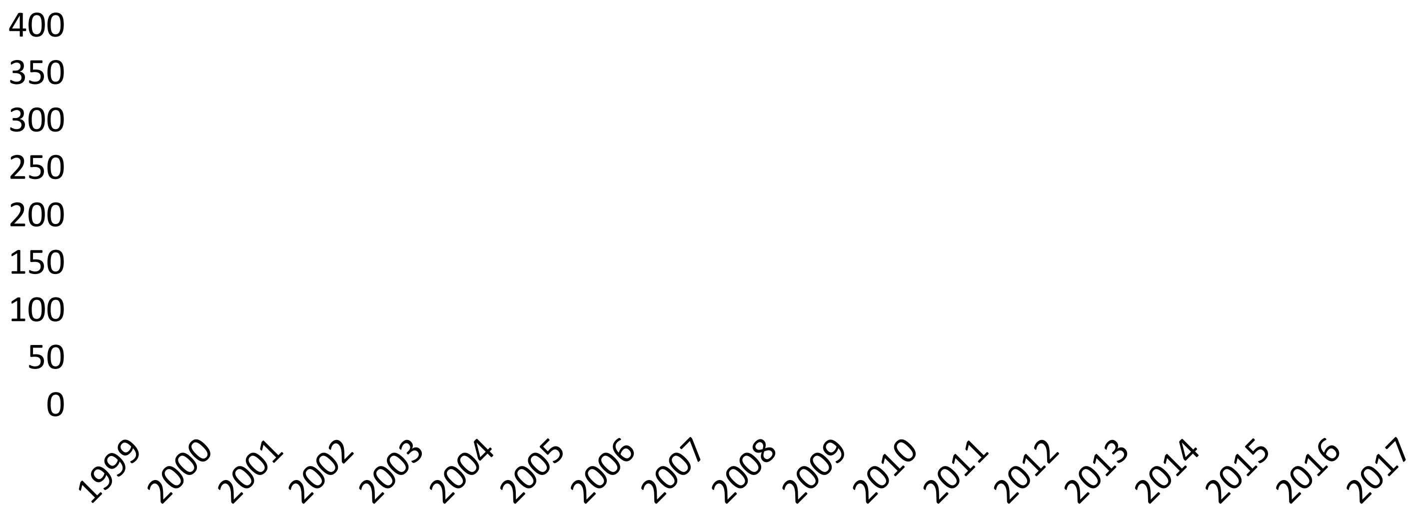
APACHE
HTTP SERVER

Inherent insecurity: Vulnerabilities and insecure designs

Implemented in unsafe languages (e.g., C/C++)

– Increasing vulnerabilities

Number of reported vulnerabilities in Linux



Data source: U.S. National Vulnerability Database

System designers prioritize performance over security

– Many insecure designs

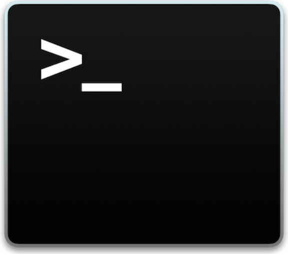
Critical system attacks exploiting vulnerabilities and insecure designs

System attacks are evolving: More and more advanced, harder and harder to defend against

Two typical goals of system attacks

To control victim systems

Control attacks



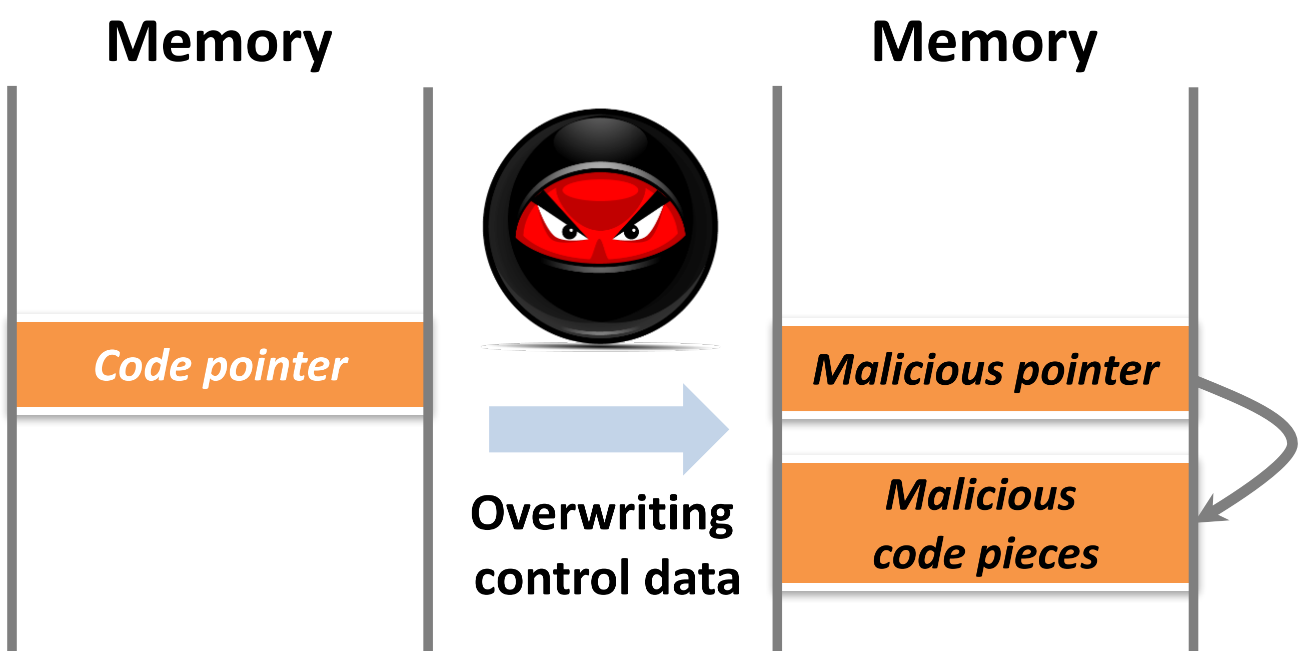
To leak sensitive data

Data leaks

Defeating both **data leaks** and **control attacks** by preventing information leaks

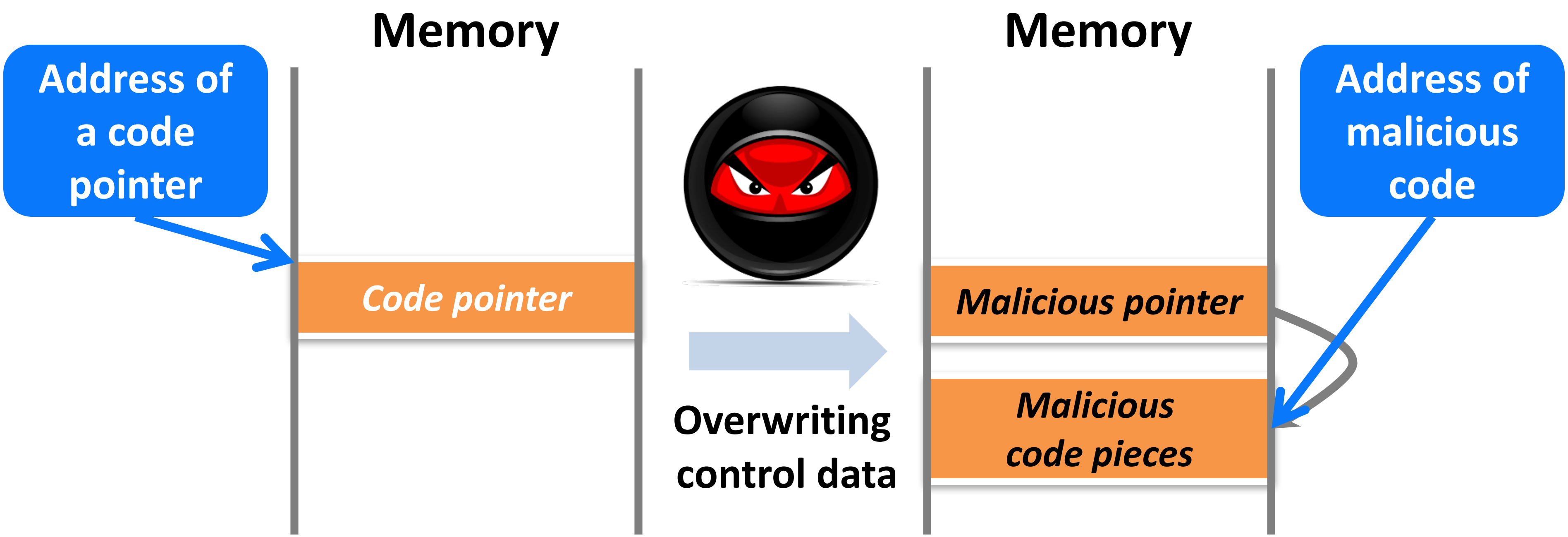
A fundamental requirement of control attacks

Attackers have to replace a code pointer with a malicious one to gain control



A fundamental requirement of control attacks

Attackers have to replace a code pointer with a malicious one to gain control

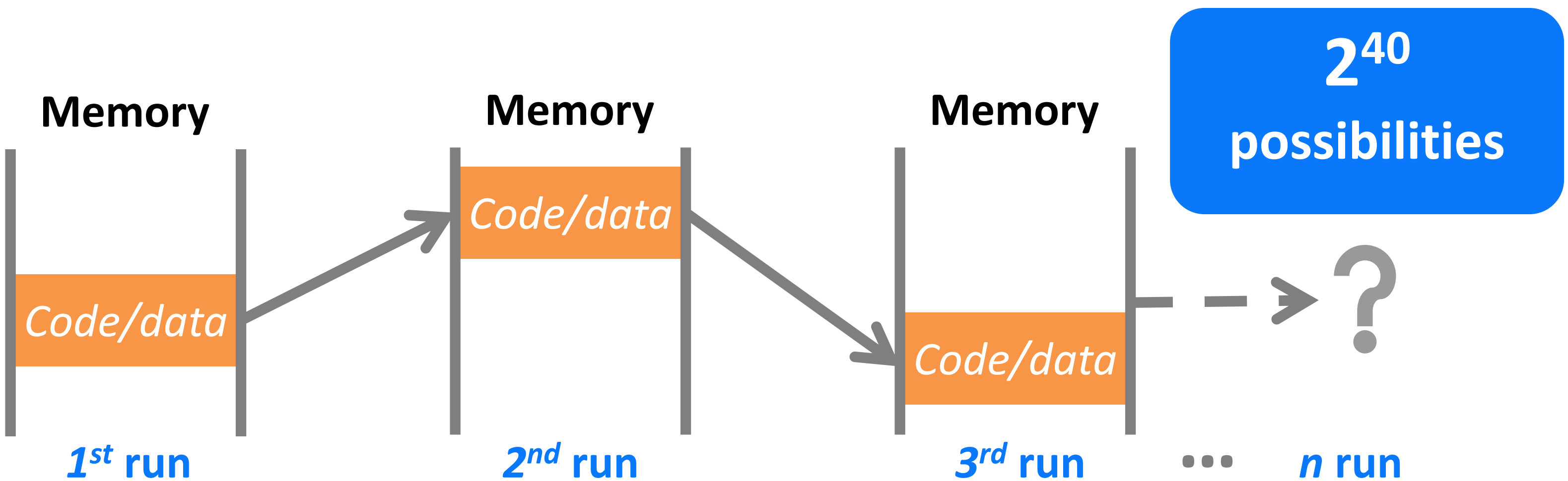


Have to know the addresses of both a code pointer and malicious code

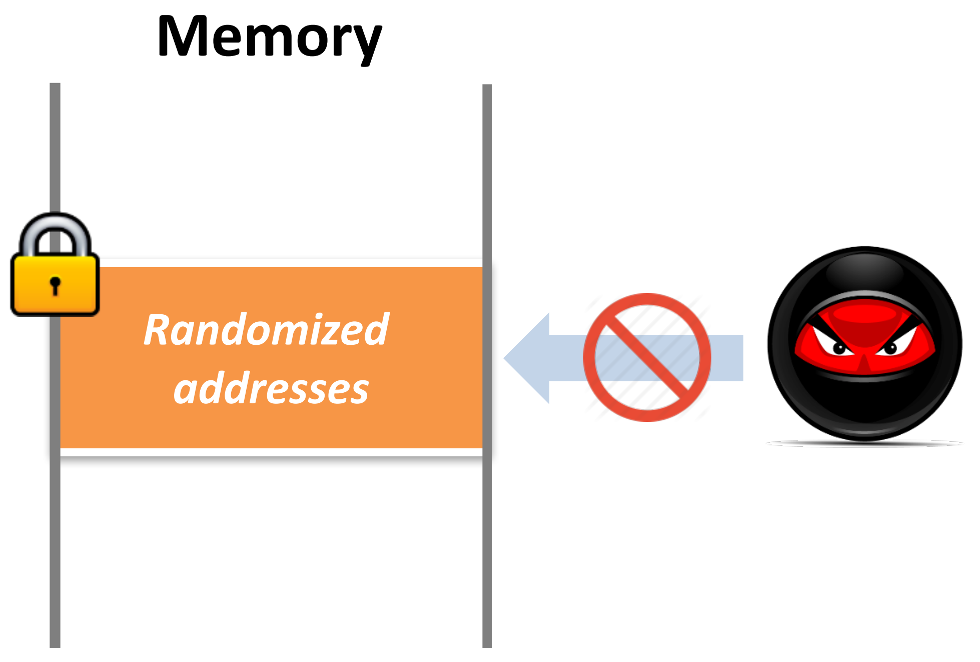
A widely deployed defense---ASLR

ASLR: Address Space Layout Randomization

– Preventing attackers from knowing addresses

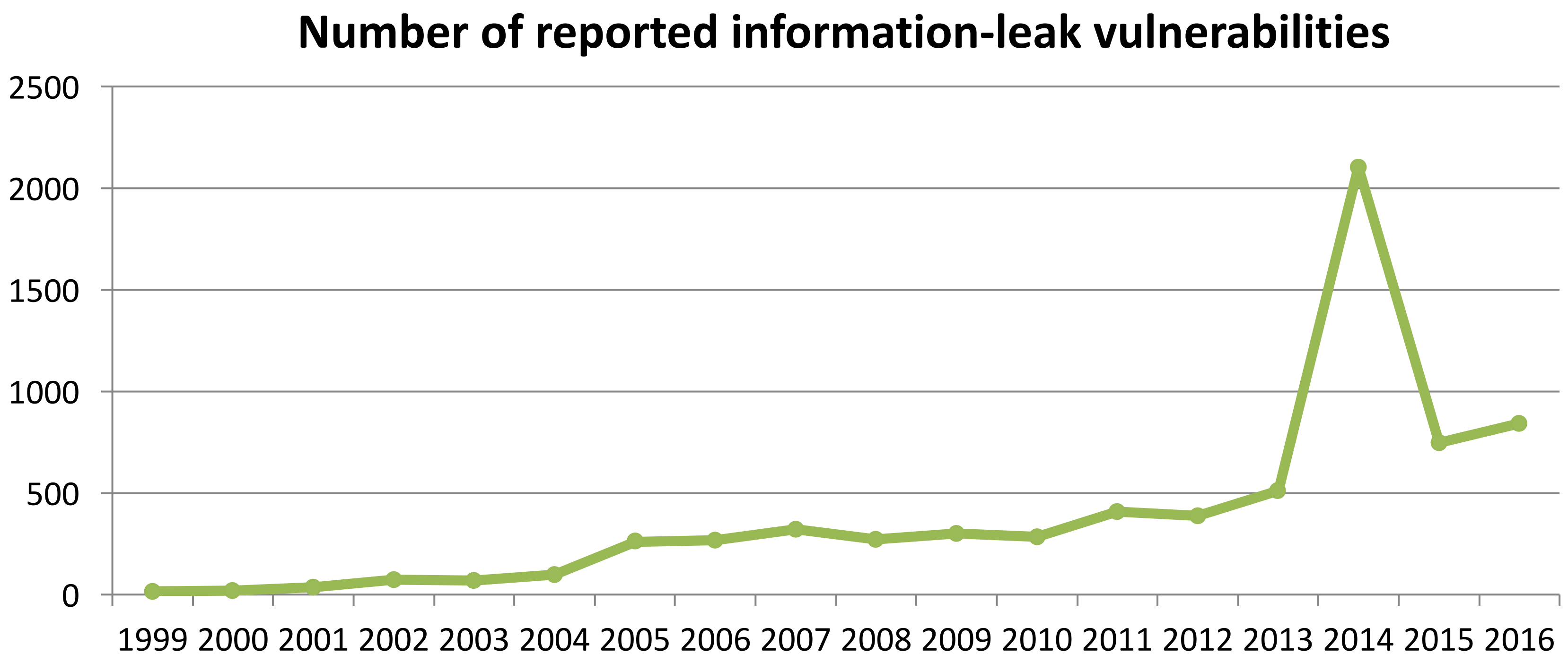


In principle, ASLR is "perfect"



ASLR is efficient, easy to deploy, and effective as long as there is no information leak

In practice, ASLR is weak



Data source: U.S. National Vulnerability Database

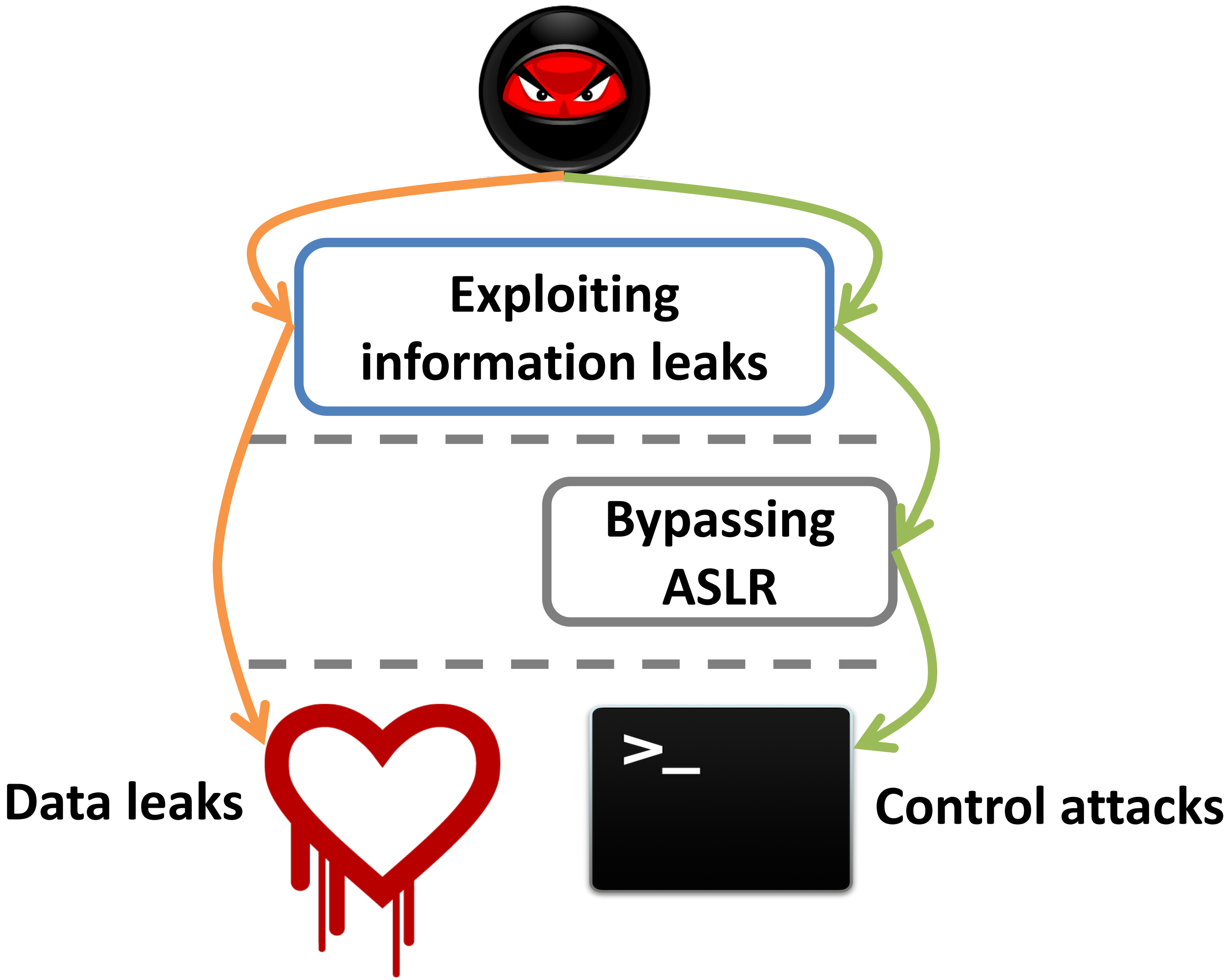
Control attacks still work because of information leaks

ASLR re-defines the prevention problem in modern systems

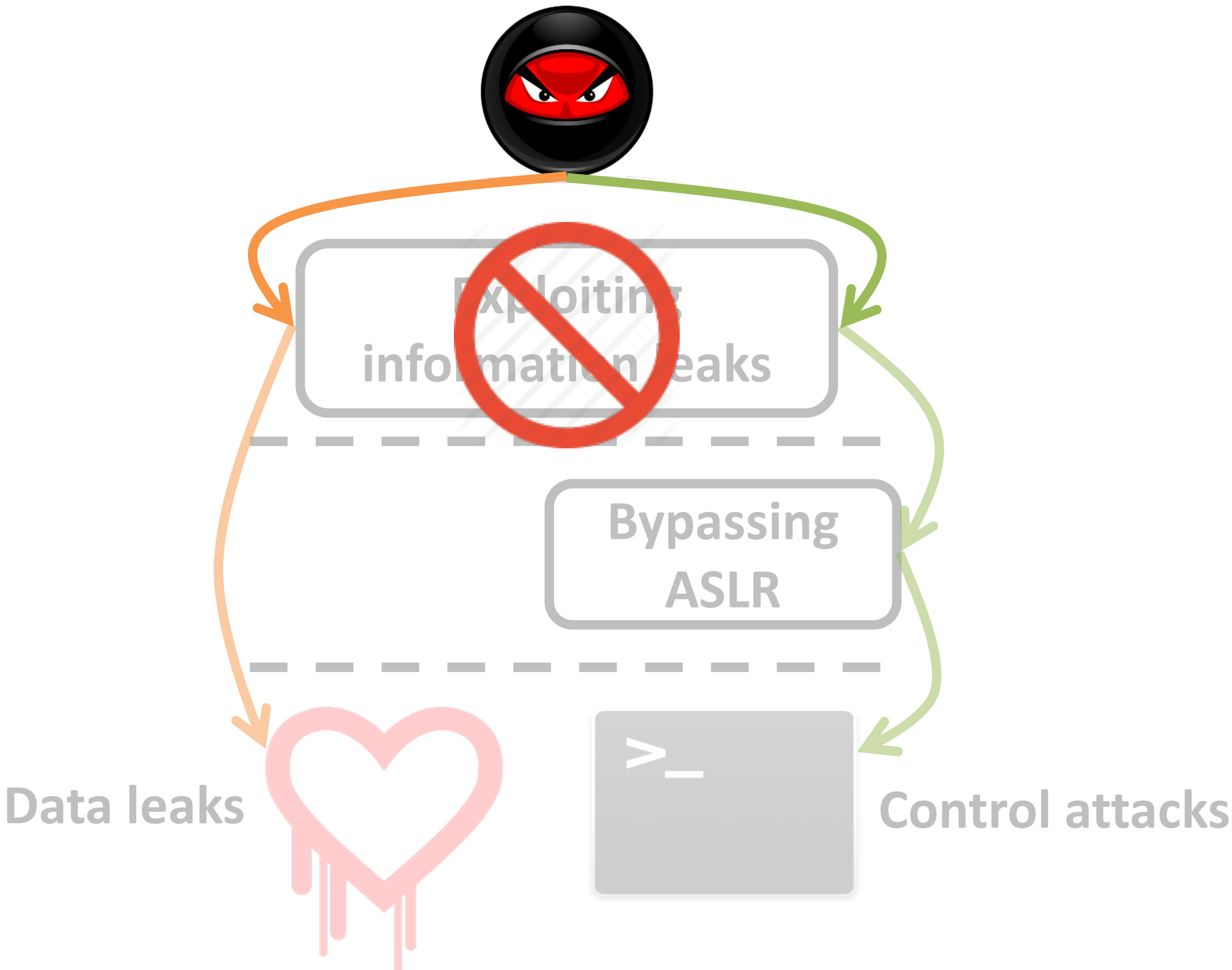
Control attack prevention problem → **ASLR** → **Address leak prevention problem**

Preventing address leaks can defeat control attacks

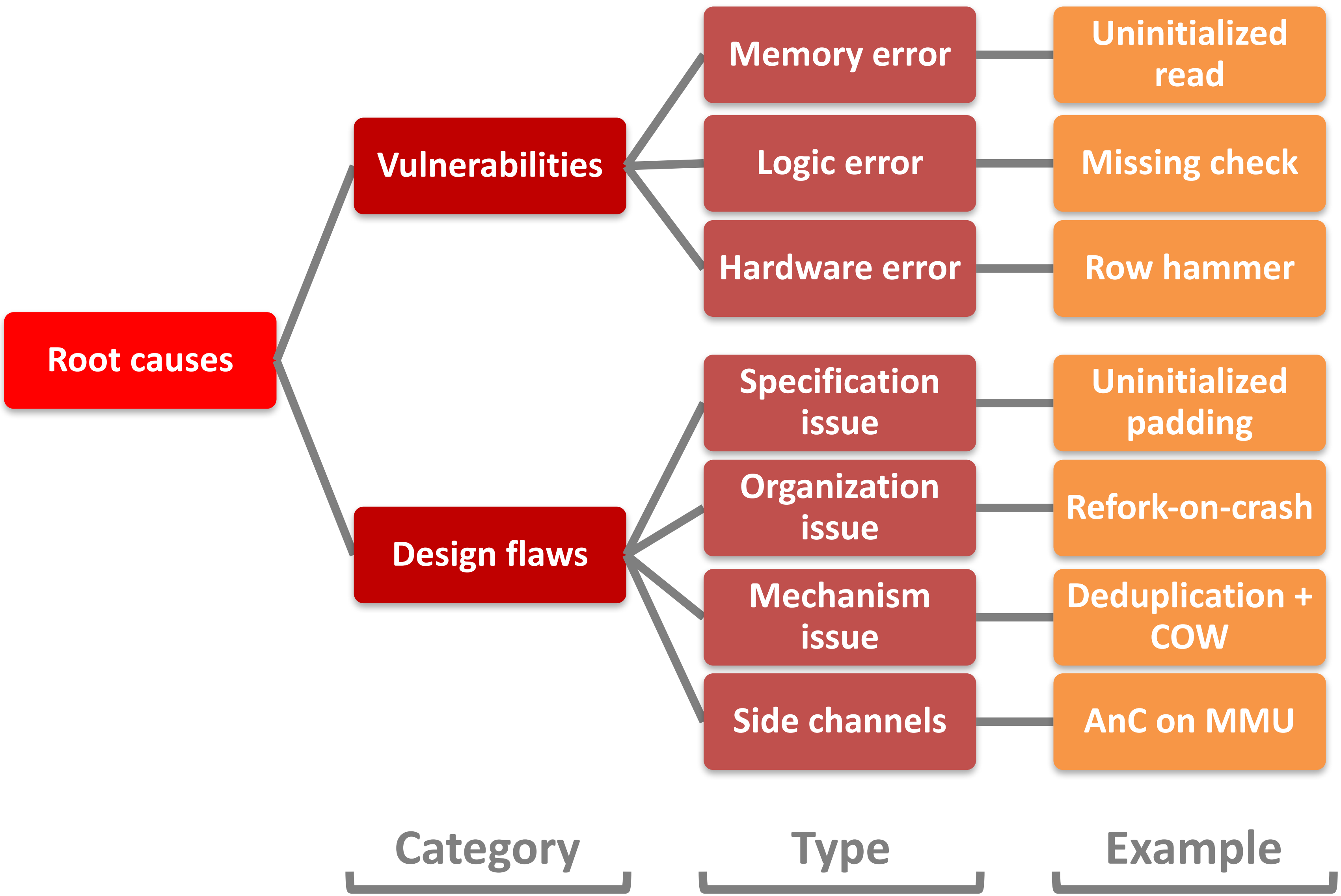
Information leak is inevitable for both attacks



Research goal:
Preventing information leaks



Root causes of known information leaks



Three ways to prevent information leaks

Eliminating information-leak vulnerabilities

- **UniSan**: Eliminating uninitialized data leaks [**CCS’16**]
- **PointSan**: Eliminating uninitialized pointers [**NDSS’17**]

Securing system designs against information leaks

- **Runtime re-randomization** for process forking [**NDSS'16**]

Protecting sensitive data from information leaks

- **ASLR-Guard**: Preventing code pointer leaks [**CCS’15**]
- **Buddy**: Detecting memory disclosures for COTS

Motivation of UniSan

OS kernels are the trusted computing base

- Contain sensitive data like crypto keys
- Deploy security mechanisms like ASLR

Hundreds of information-leak vulnerabilities

- Data leaks
- ASLR bypass

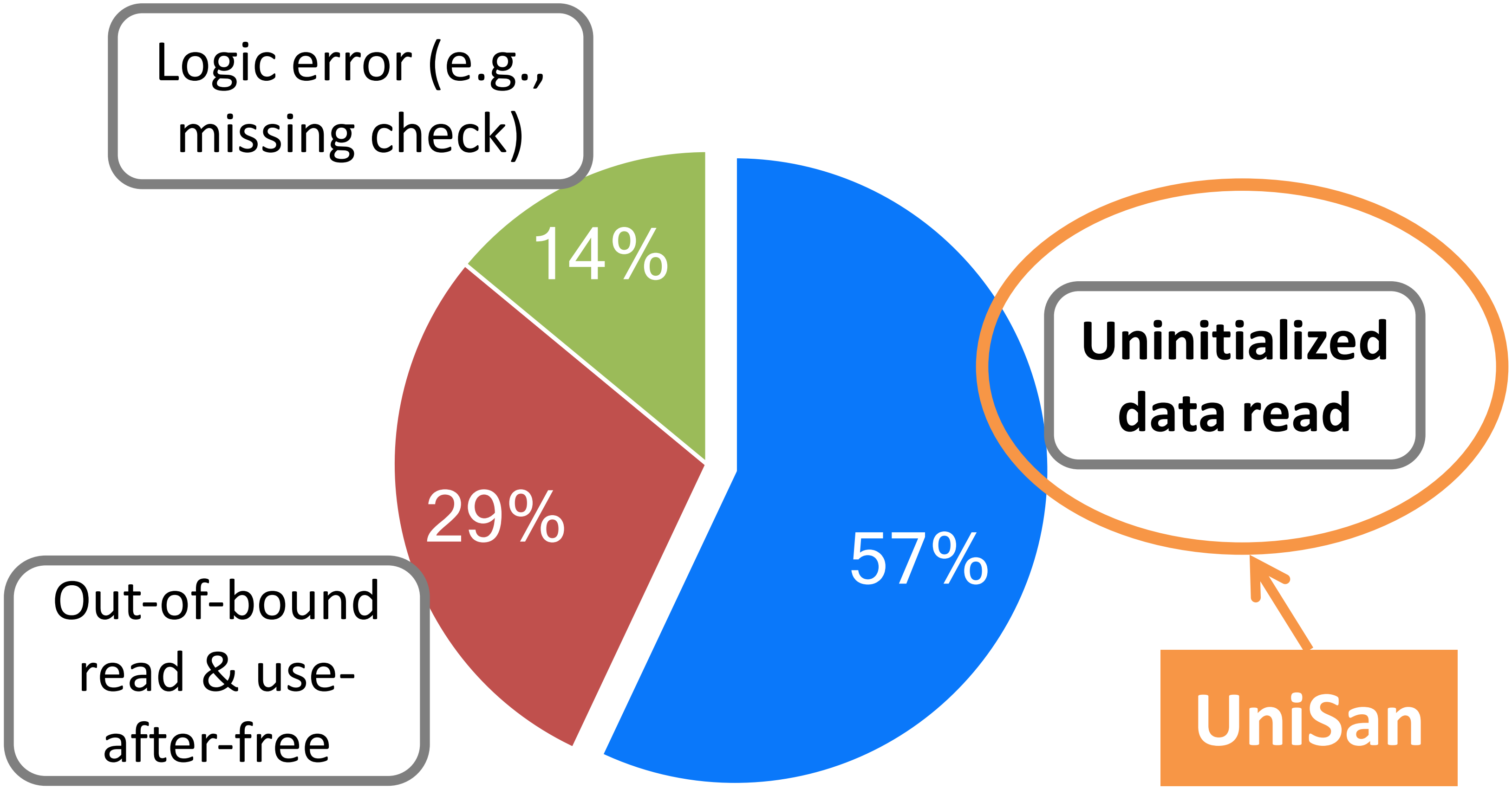
UniSan:
To eliminate (the most common) information-leak vulnerabilities in OS kernels

→ Mitigate data leaks, code-reuse and privilege-escalation attacks

Main contributions of UniSan

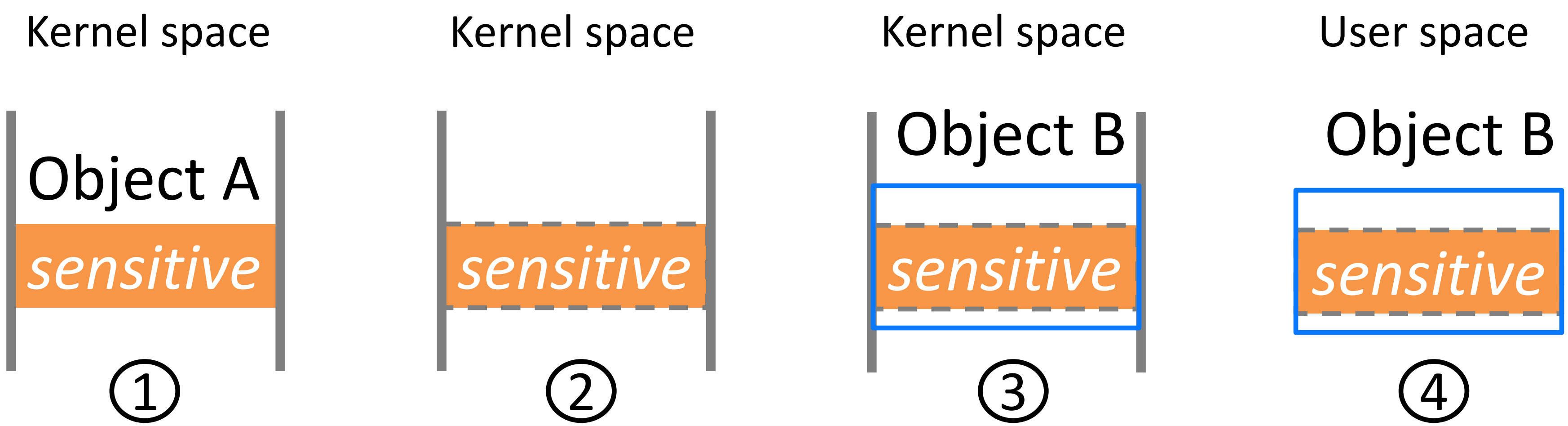
- Automatically secure the Linux and Android kernels with negligible runtime overhead
- Reported and patched **19** kernel vulnerabilities
  - CVE-2016-5243, CVE-2016-5244, CVE- 2016-4569, CVE-2016-4578, CVE-2016-4569, CVE-2016-4485, CVE-2016-4486, CVE-2016-4482, ……
- Found and fixed a critical security problem in compilers
- Porting UniSan to GCC for adoption

The main cause of information leaks: Uninitialized data read



Data source: U.S. National Vulnerability Database
(kernel information leaks reported between 2013 and 2016)

How an uninitialized data read leads to an information leak



User A
obje
writes “*sensitive*”
into it

“*sensitive*” is not
cleared

initialization;
“*sensitive*” kept

eads
t B;
“*sensitive*”
leaked!

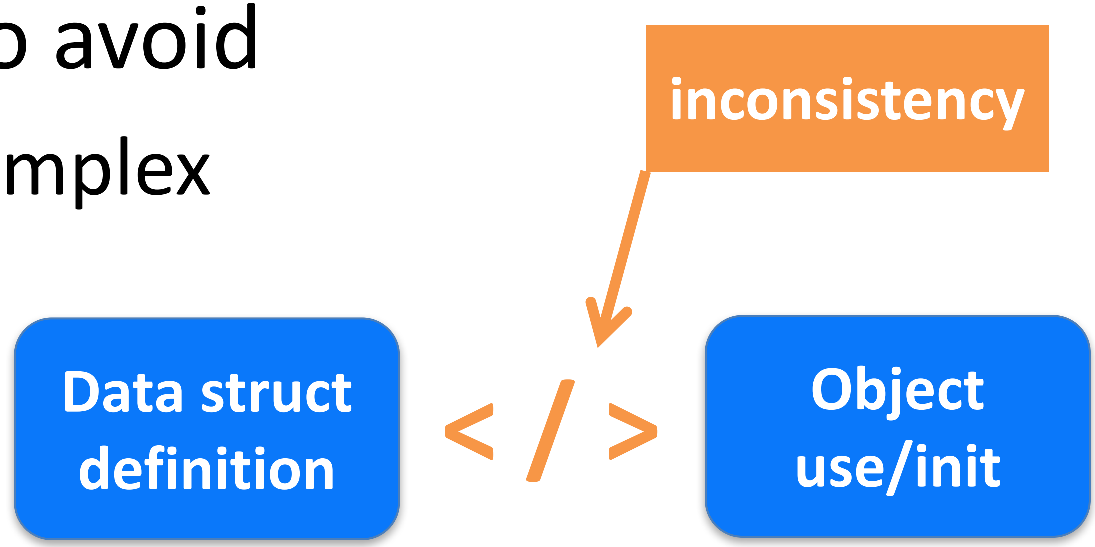
We call such information leaks
“uninitialized data leaks”

Troublemaker: Developer

Missing field initialization: Blame the developers?

Difficult to avoid

- Too complex



Troublemaker: Compiler

Data structure padding: A fundamental feature for improving CPU efficiency

```
struct test {
    un
    un
   /*
};
```

```
/* both fields (5 bytes) are initialized*/
```

A critical and prevalent security problem: Programs are built by compilers!

```
/* leaking uninitialized 3-byte padding*/
copy_to_user(dest, &t, sizeof(t));
```

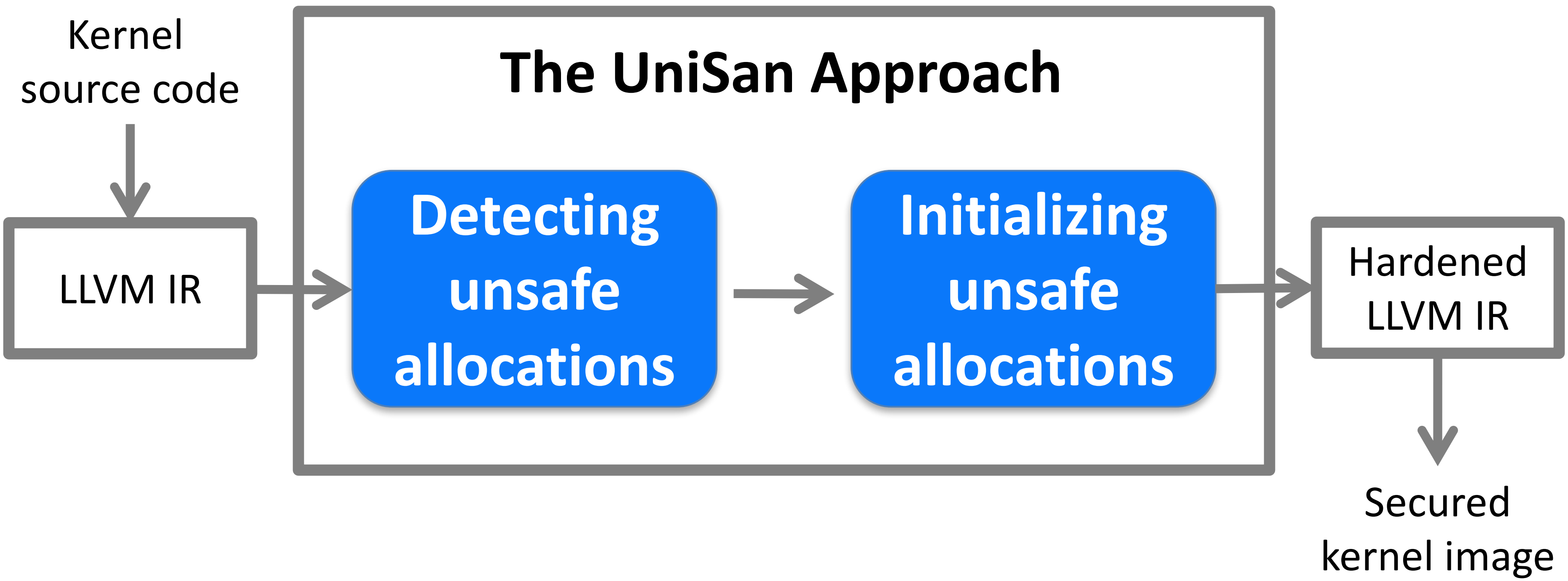
The root cause: C specifications (C11)

Chapter §6.2.6.1/6

"When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values."

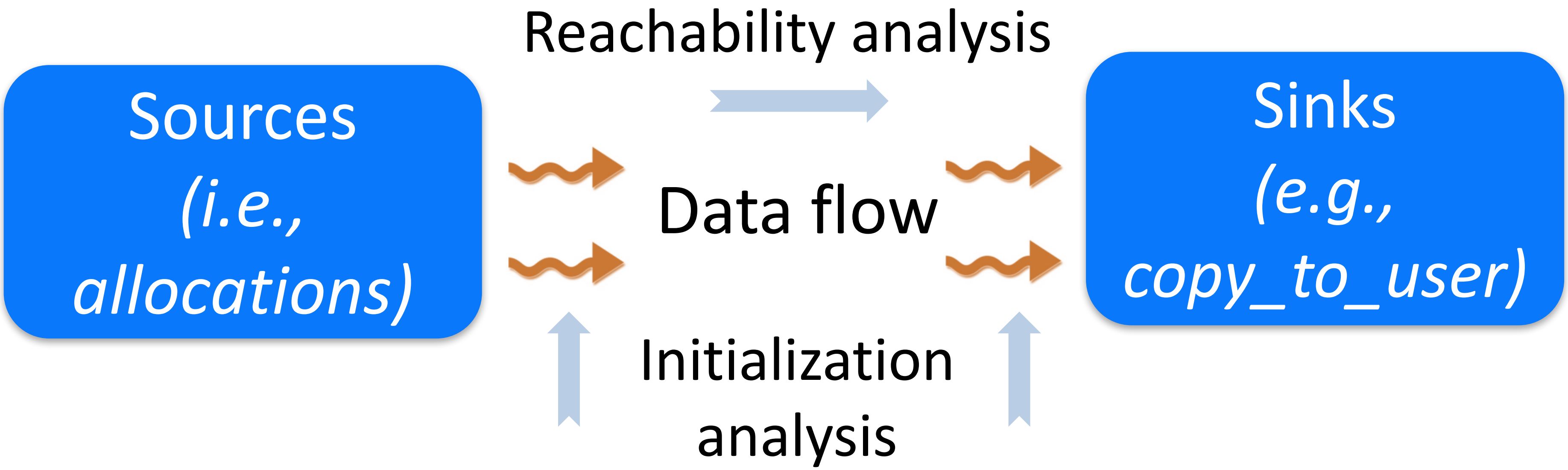
UniSan: A compiler-based solution

Simply initialize all allocated objects? Too expensive!



Unsafe allocation detection

Byte-level and flow-, context-, and field-sensitive taint tracking



Technical challenges in detection

- Global call-graph construction
  - Conservative type analysis for indirect calls
- Byte-level tracking
  - Maintaining offsets of fields
- Eliminating false negatives

Be conservative!
Assume it is unsafe for unhandled special cases!

Zero-initializing all unsafe allocations

Stack	Heap
obj = 0	kmalloc(size, flags|__GFP_ZERO)
memset(obj, 0, sizeof(obj))	

Zero initialization is **semantic preserving**

- Robust
- Tolerant of false positives

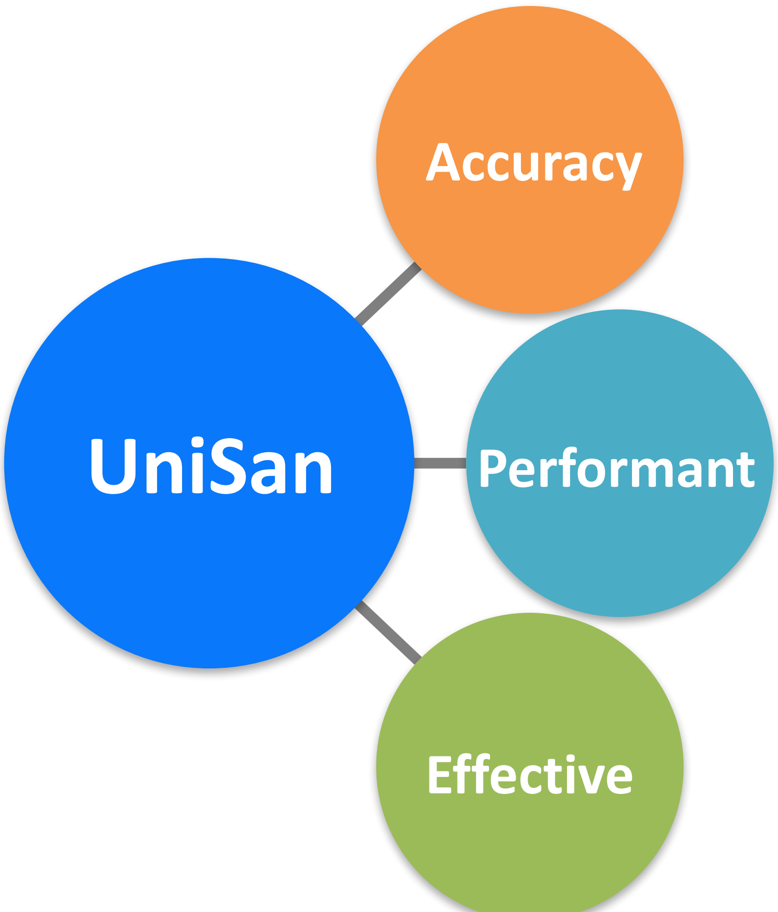
LLVM-based implementation

An analysis pass + an instrumentation pass

How to use UniSan: *$ unisan @bitcode.list*

UniSan is performant and effective

Applied to the latest Linux kernel and Android kernel



10% (2K) of allocations are detected as unsafe

Negligible runtime overhead:

- System operations: **1.36%**
- Web servers: **<0.1%**
- User programs: **0.54%**

Prevented known and new vulnerabilities

- **19** have been confirmed and fixed by Google and Linux

Three ways to prevent information leaks

Eliminating information-leak vulnerabilities

- **UniSan**: Eliminating uninitialized data leaks [**CCS'16**]
- **PointSan**: Eliminating uninitialized pointers [**NDSS'17**]

Securing system designs against information leaks

- **Runtime re-randomization** for process forking [**NDSS'16**]

Protecting sensitive data from information leaks

- **ASLR-Guard**: Preventing code pointer leaks [**CCS'15**]
- **Buddy**: Detecting memory disclosures for COTS

Three ways to prevent information leaks

Eliminating information-leak vulnerabilities

- **UniSan**: Eliminating uninitialized data leaks [**CCS'16**]
- **PointSan**: Eliminating uninitialized pointers [**NDSS'17**]

Securing system designs against information leaks

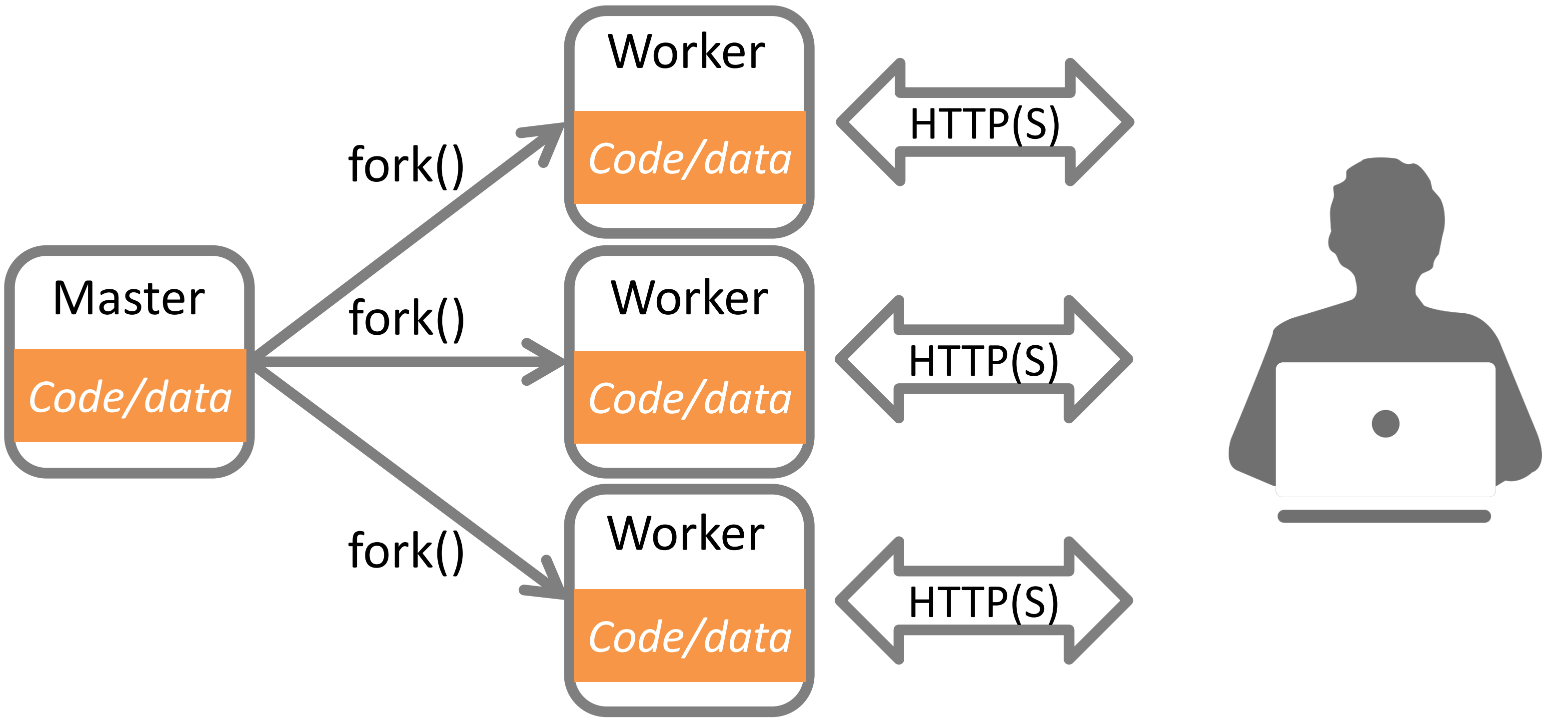
- **Runtime re-randomization** for process forking [**NDSS'16**]

Protecting sensitive data from information leaks

- **ASLR-Guard**: Preventing code pointer leaks [**CCS'15**]
- **Buddy**: Detecting memory disclosures for COTS

The insecure process forking violates ASLR

A common design of web servers:



Exactly same memory layout. Re-fork upon worker crashes

The clone-probing attack

Attack goal: To guess sensitive data (say randomized return address) with a simple buffer overflow

Stack of a web server

Buffer overflow	return address 12 34 56 78 9a bc ed f0	
AAAAAAA	00 34 56 78 9a bc ed f0	Crash, try another one
…	…	
AAAAAAA	12 34 56 78 9a bc ed f0	Finally, get all bytes

Brute-forcing complexity is reduced from 2^{64} to $8*2^{8}$

Usually can be done within two minutes.

Re-randomizing the memory layout of forked processes

Automatic pointer tracking → Memory layout re-randomization → Dangling pointer patching

Main contributions

- A new mechanism for automatic pointer tracking at runtime (using Intel's Pin)
- Successfully applied it to Nginx web server

Three ways to prevent information leaks

Eliminating information-leak vulnerabilities

- **UniSan**: Eliminating uninitialized data leaks [**CCS'16**]
- **PointSan**: Eliminating uninitialized pointers [**NDSS'17**]

Securing system designs against information leaks

- **Runtime re-randomization** for process forking [**NDSS'16**]

Protecting sensitive data from information leaks

- **ASLR-Guard**: Preventing code pointer leaks [**CCS'15**]
- **Buddy**: Detecting memory disclosures for COTS

Motivation of ASLR-Guard

Code-reuse attacks are rampant and critical

Leaking a code pointer to first bypass ASLR has become a **prerequisite** for code-reuse attacks

ASLR-Guard:
To prevent code-pointer leaks to defeat code-reuse attacks

(a user-space security mechanism against remote attackers)

Two main contributions

A systematic way of discovering code pointers

Two techniques of preventing code pointer leaks

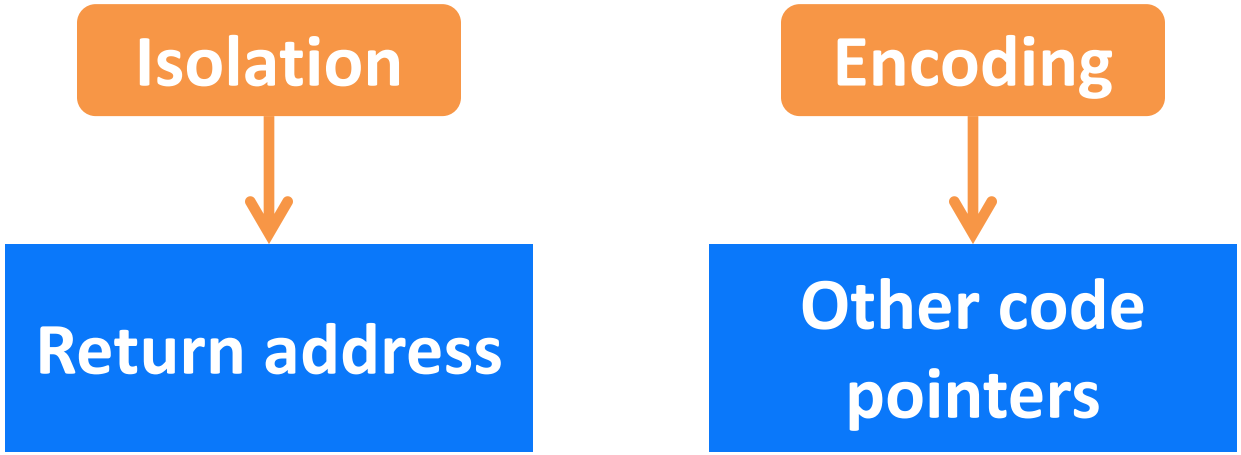
Empirical code pointer discovery

By relocation

Lesson: Code pointer discovery is practical; programs built by modern compilers create code pointers regularly

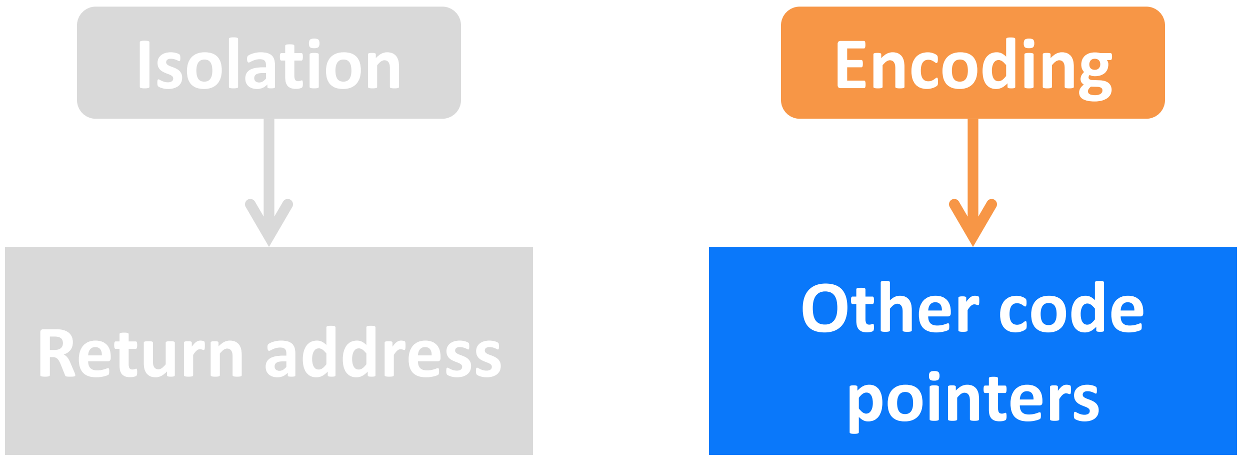
How are code pointers created?

Isolating or encoding code pointers



- Function pointer
- Entry pointer
- Signal handler
- …

Isolating or encoding code pointers



- Function pointer
- Entry pointer
- Signal handler
- …

Encoding code pointers

When isolation is hard

Three requirements for encoding

- **Confidentiality**: Cannot crack
- **Integrity**: Cannot modify
- **Efficiency**: Be performant

Fast code pointer encoding

```c
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax

Fast code pointer encoding

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax

%gs

Random Mapping Table (in safe region)

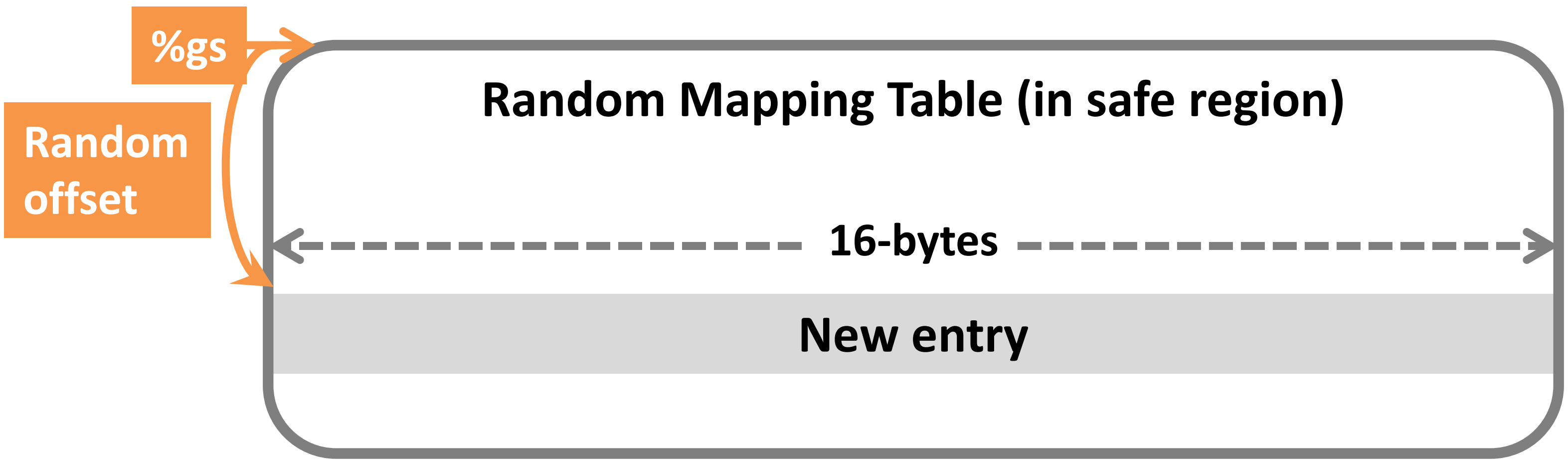
Mapping entries…

Fast code pointer encoding

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



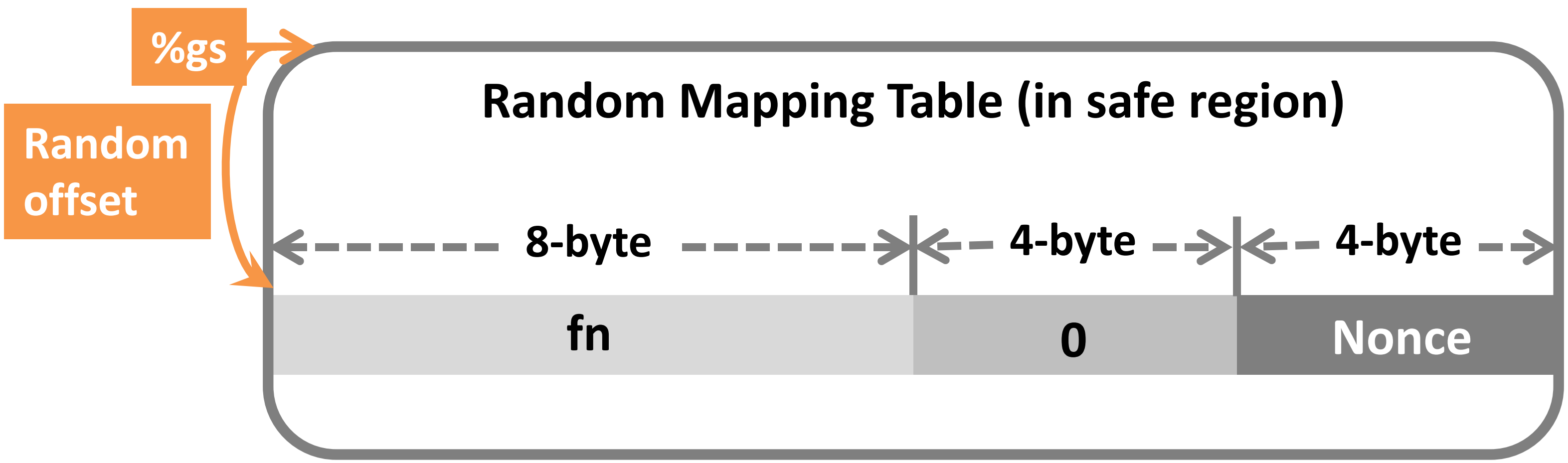
Step1: create an entry with a random offset

Fast code pointer encoding

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



Step1: create an entry with a random offset

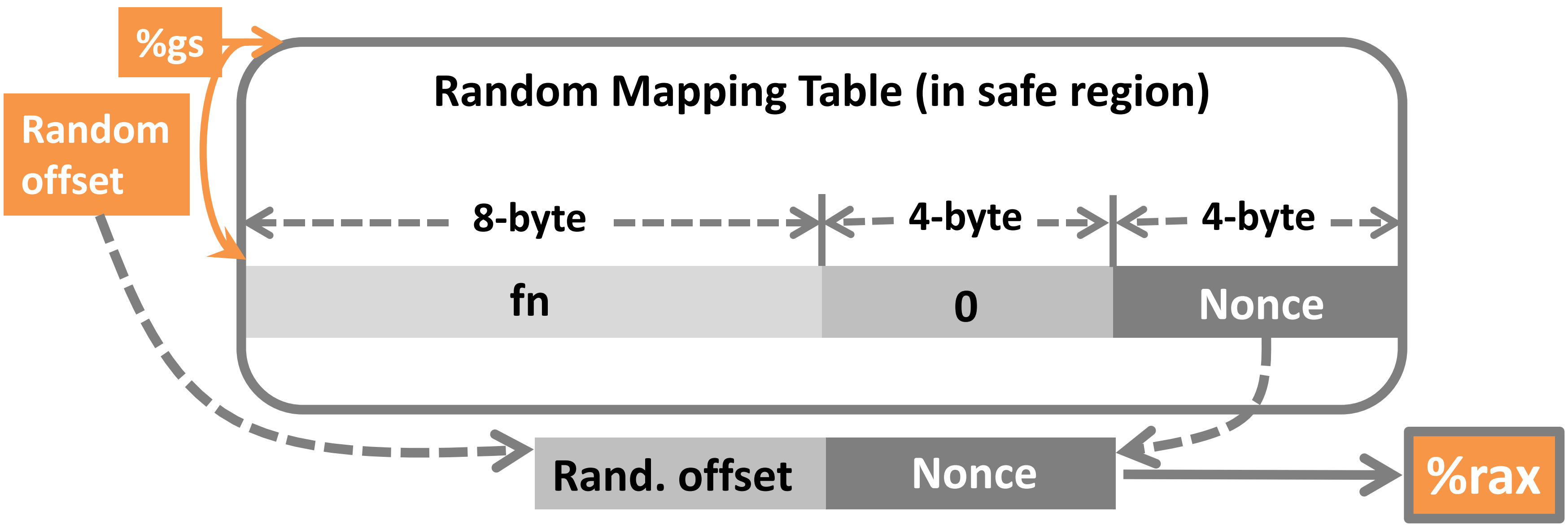
Step2: save *fn* in first 8-byte, then 4-byte 0 and 4-byte nonce

Fast code pointer encoding

```
void hello();
void (*fn)() = hello;
```

Assembly:

lea 0x1234(%rip), %rax



Step1: create an entry with a random offset

Step2: save *fn* in first 8-byte, then 4-byte 0 and 4-byte nonce

Step3: save the 4-byte random offset and nonce into %rax

Extremely fast decoding

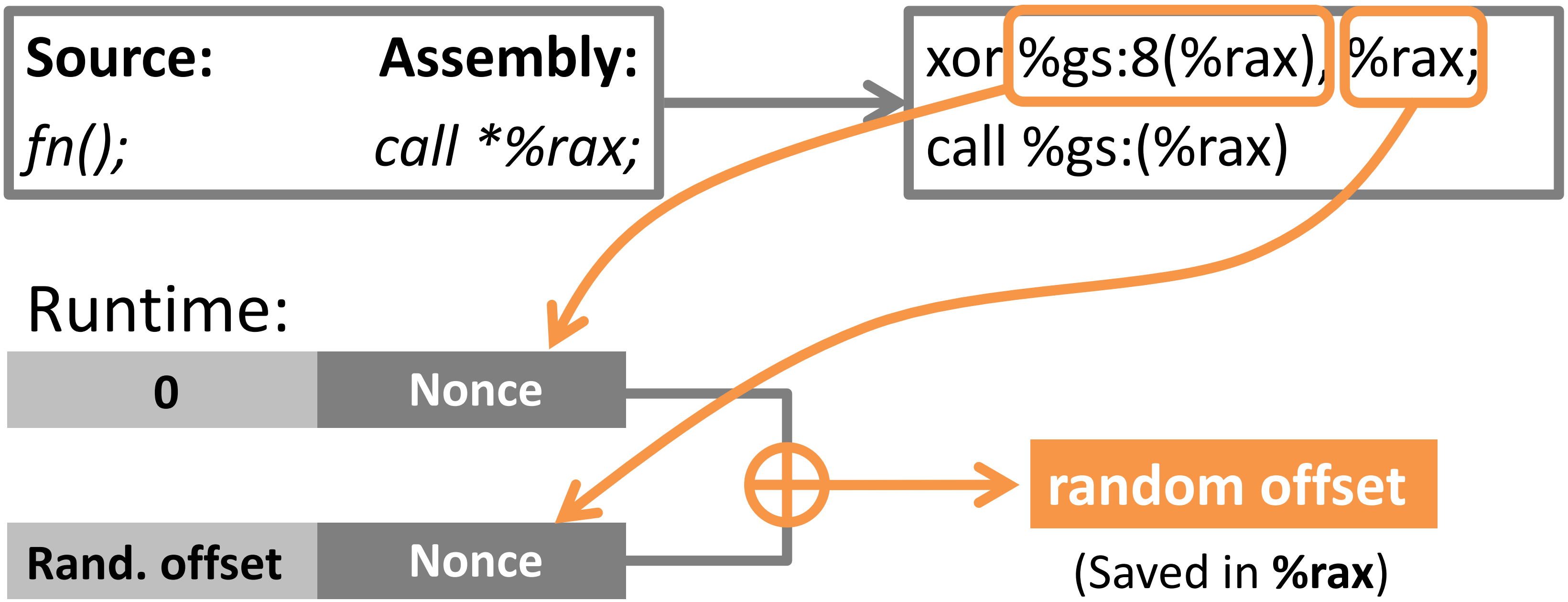
Compile time:

Source:	**Assembly:**
fn();	*call *%rax;*

```
xor %gs:8(%rax), %rax;
call %gs:(%rax)
```

Extremely fast decoding

Compile time:



Extremely fast decoding

Compile time:

Source:	**Assembly:**
fn();	*call *%rax;*

xor %gs:8(%rax), %rax;
call %gs:(%rax)

Runtime:

0	**Nonce**
Rand. offset	**Nonce**

$\oplus$ → **random offset**

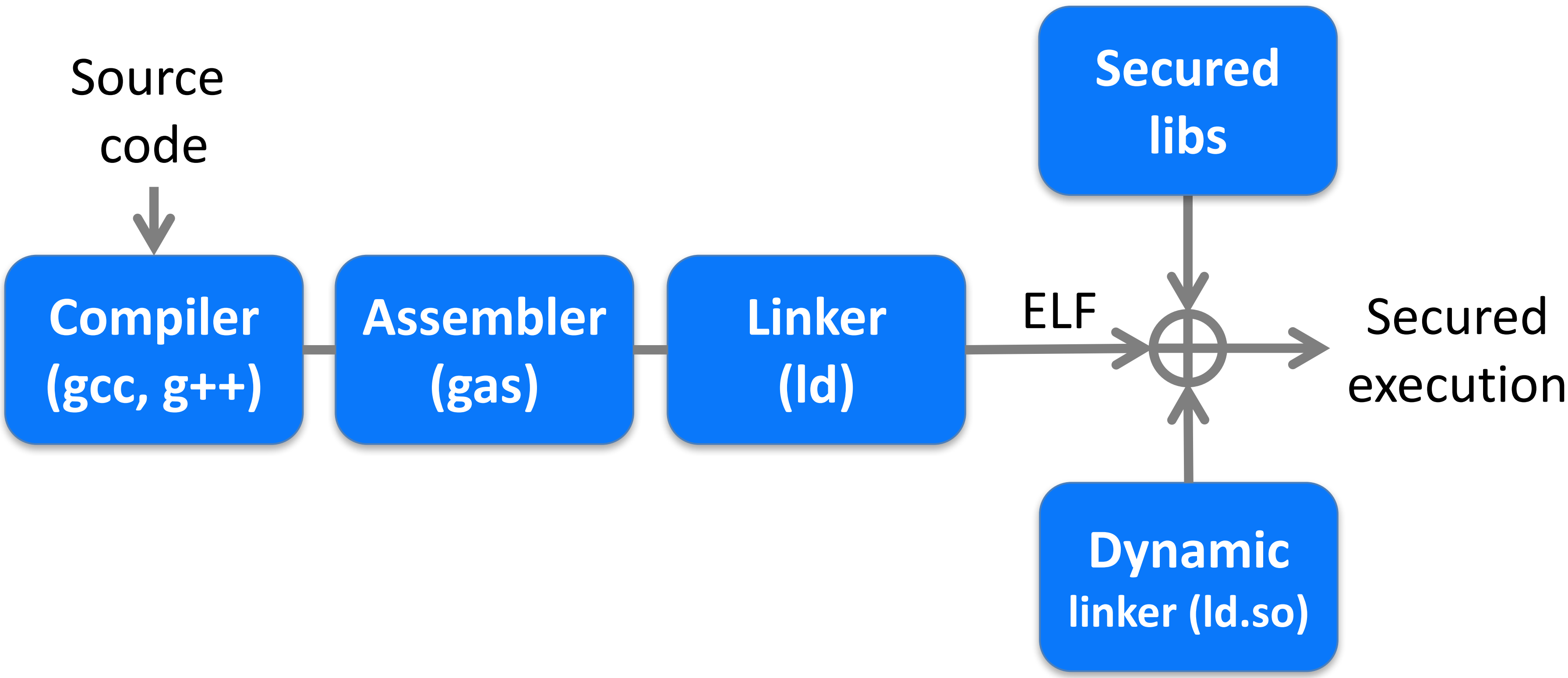
(Saved in **%rax**)

%gs:(%rax) points to ***"fn"*** in random mapping table,
so, call %gs:(%rax) → call fn

Extremely fast decoding

Extremely efficient decoding: Only **one XOR** operation!

ASLR-Guard: A toolchain and a runtime



GNU toolchain
(~3,000 LoC changes)

C/C++ Runtime

ASLR-Guard is performant and effective

Applied to the SPEC Benchmarks and the Nginx web server

Negligible performance overhead:

- **Runtime overhead: <1%**
- Binary size increase: **6%**
- Memory overhead: **2MB**

Attacks such as BlindROP are defeated!

Three ways to prevent information leaks

Eliminating information-leak vulnerabilities

- **UniSan**: Eliminating uninitialized data leaks [**CCS’16**]
- **PointSan**: Eliminating uninitialized pointers [**NDSS’17**]

Securing system designs against information leaks

- **Runtime re-randomization** for process forking [**NDSS'16**]

Protecting sensitive data from information leaks

- **ASLR-Guard**: Preventing code pointer leaks [**CCS’15**]
- **Buddy**: Detecting memory disclosures for COTS

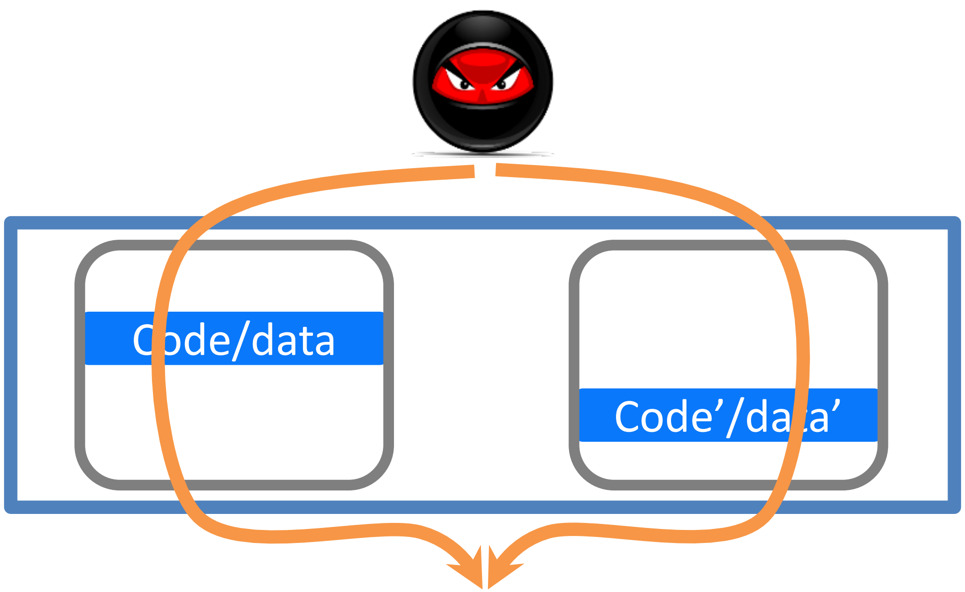
Motivation of Buddy

- Memory disclosures are **critical**
  - Data leaks
  - Defense mechanism bypass
- Memory disclosures are **common**
  - Thousands of vulnerabilities each year, still increasing
- Memory disclosures are **diverse**
  - Various causes
  - Various memory data types
- Memory disclosure prevention is **expensive**
  - Much more expensive than preventing invalid write

How to stop memory disclosures in a **general** and **practical** manner?

Buddy: An replicated execution-based approach

Seamlessly maintain two identical processes with diversified data/layout (same semantics)



Compare outputs: disclosures will cause divergences

A formal model for Buddy

- Detecting points such as I/O write
  - *0, 1,…i*
- States at detecting point i
  - Original process: $S_{o,i}$,
  - Buddy instances: $<S_i, S_i'>$
- Mapping buddy states to original state
  - Mapping function: $Map(S_i) = Map(S_i') = S_{o,i}$
- Transition functions for all processes
  - Take a state S_i and an input I, and produce next state
  - $T(S_i, I) = S_{i+1}$; same for $T'()$ and $T_o()$

Two properties of Buddy

Equivalence property

- Buddy must preserve semantics for original process under normal execution

(1). $Map(S_0) = Map(S_0') = S_{o,0}$

(2). $\forall\, 0 \leq i \leq N,\ \forall\, I \in \text{Normal inputs:}$
$Map(T(S_i, I)) = Map(T'(S_i', I)) = T_o(S_{o,i}, I)$

Divergence property

- Buddy must detect divergences when memory disclosures occur

(3). $\forall\, 0 \leq i \leq N,\ \forall\, I \in \text{Inputs:}$
$T(S_i, I))$ or $T'(S_i', I)) \in \text{Memory disclosures}$
$\Rightarrow Map(S_{i+1}) \neq Map(S_{i+1})$

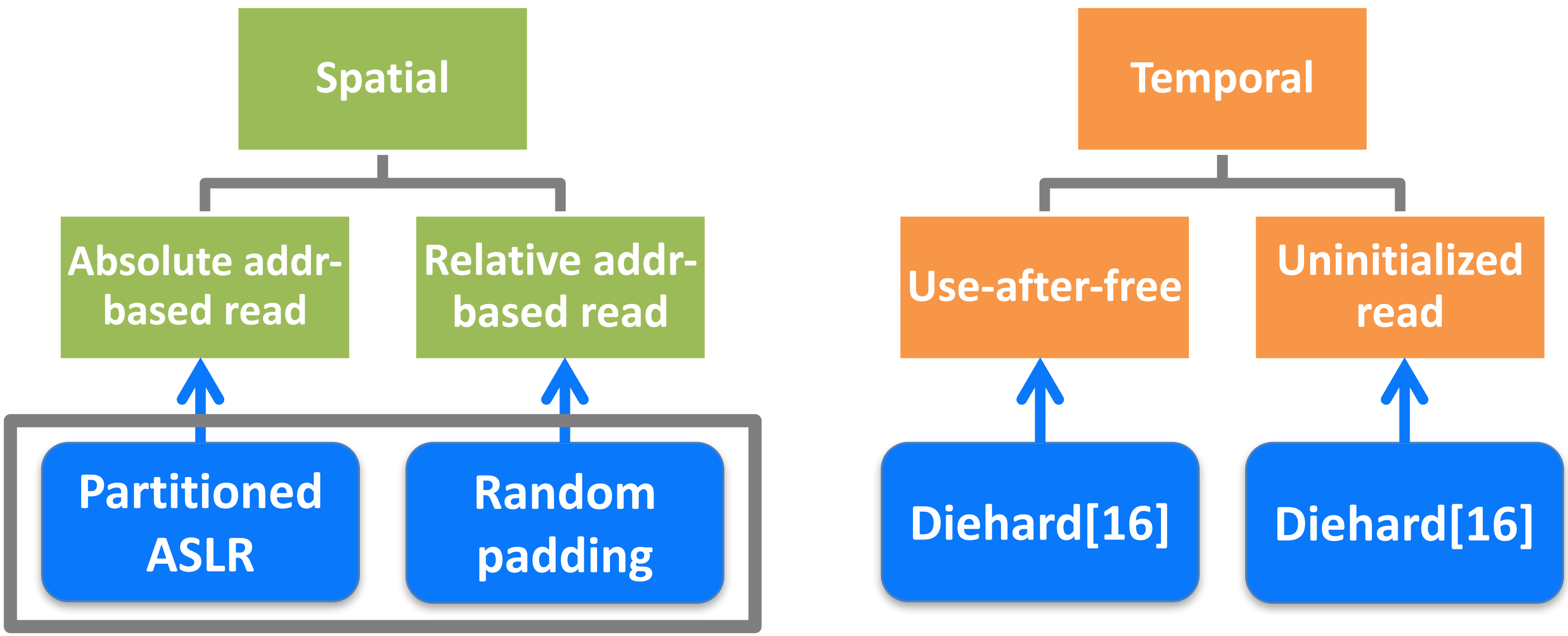
Assumptions of Buddy

- Memory disclosures go through pre-defined detecting points
- Programs do not intentionally use unspecified memory
- We have the list of non-determinism sources
- We have a multi-core CPU

Detecting memory disclosures with Buddy

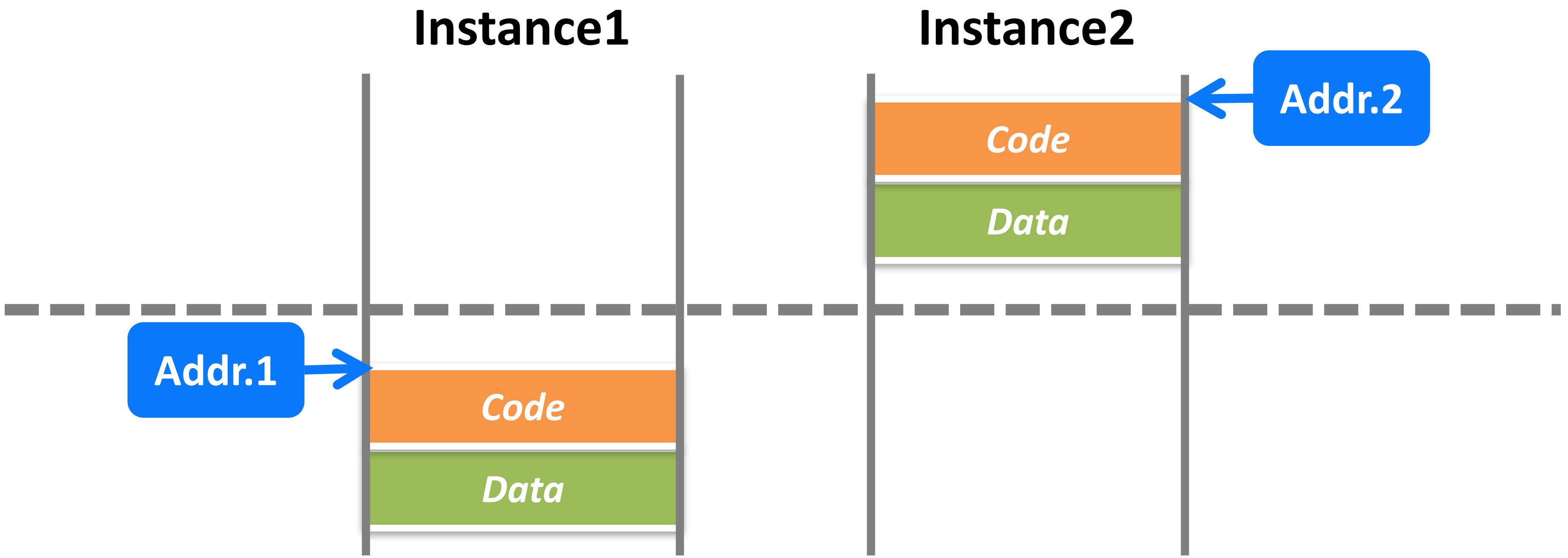
A general replicated execution framework

Two new schemes built upon Buddy



Partitioned ASLR

- Detect absolute address-based over-reads
- Partition address space into two sub-spaces
- Enable randomization for each sub-space
  - Apply PIC and modify loader (ld.so)



Properties of partitioned ASLR

Equivalence property – **Yes**

- PIC and ASLR are non-interference
- No change to semantics

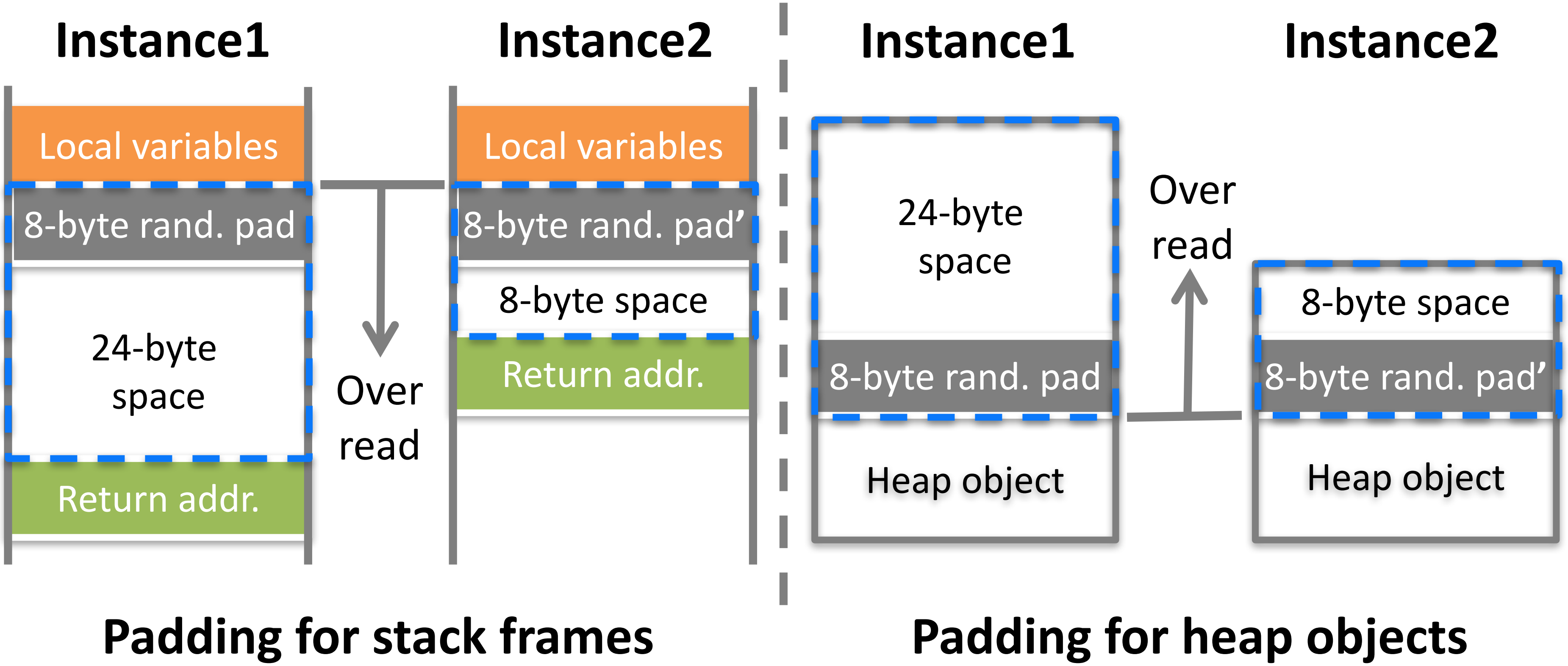
Divergence property – **Yes**

- Sub-spaces are non-overlapping: Addr1 ≠ Addr2
- Any absolute addr-based over-read will always result in one instance crashing

Random padding

Detect relative address-based over-reads

Paddings have different values and sizes



Properties of random padding

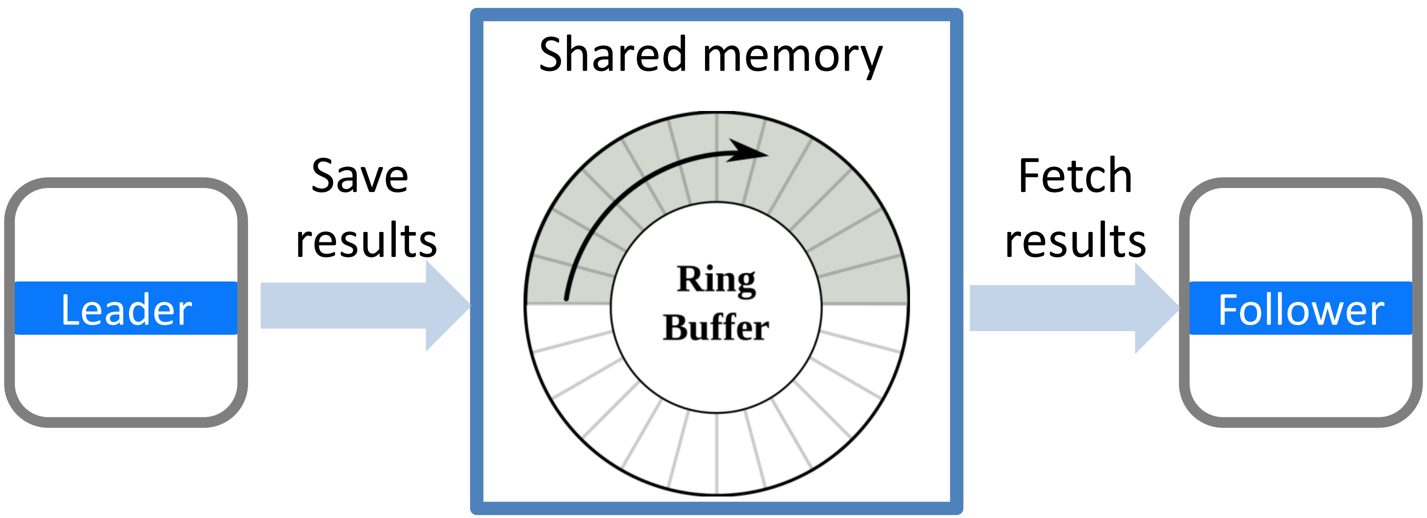
- **Equivalence** property – **Yes**
  - Rearrange memory layout of object
  - No change to semantics (assuming semantics do not depend on object memory layout)
- **Divergence** property
  - Continuous reads – **Yes**
    - Paddings have different values
  - Offset-based reads
    - If target data is random – **(2^N-1)/2^N, where N = read bits**
    - If target data has a layout pattern, e.g., repeating – **Probabilistic**

Efficient coordination of Buddy instances

Virtualizing points and interception

- Most system calls -- syscall table patching
- All virtual system calls -- GOTPLT table patching
- RDTSC and RDRAND instructions -- Binary rewriting

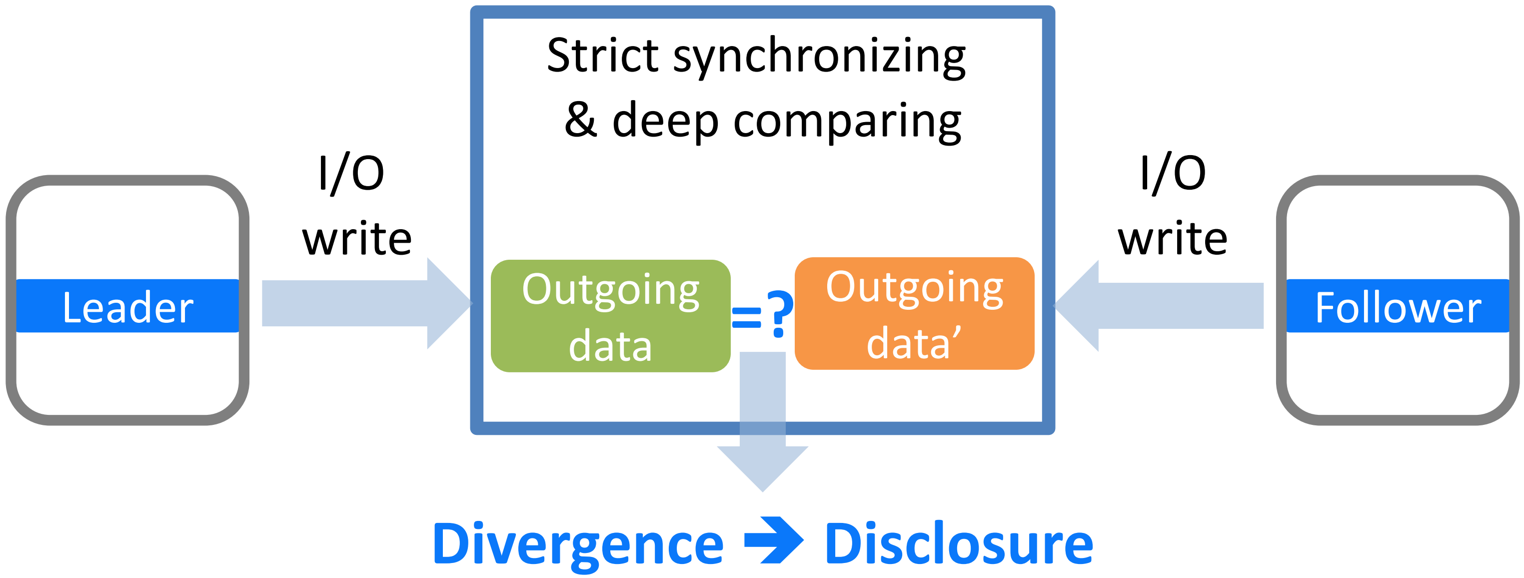
Ring buffer-based coordination



Single-point synchronization and detection

Detecting at only socket write and file write

Crashing is directly treated as a divergence



Extensive evaluation

- Testing programs
  - SPEC bench programs, Apache server, Nginx server, Lighttpd, PHP, and OpenSSL
- Experimental setup
  - Eight-core machine with 64-bit Linux
- Evaluation scope
  - Robustness
  - Security
  - Performance

Evaluation of robustness and security

- Robustness: Extensive empirical testing
  - No error/crash observed
  - Outputs with and without Buddy are the same
  - One false positive---use of uninitialized memory
- Security: Real attacks detected
  - Data-oriented exploits[82]
  - BlindROP[20]
  - Loop timing-based leaks[156] (absolute-addr-based read)
  - Heartbleed attack[61]

Evaluation of performance

- SPEC Benchmarks
  - Light-load CPU: 2.34%
  - Heavy-load (99% usage) CPU: 8.3%
- Web Benchmarks
  - Concurrency 1-256, Worker 1-8
    - 0%-10.8% with geo-mean 3.6%
  - File size 1KB-16MB (with c=16 and p=4)
    - 1.4%-8.7% with geo-mean 4.6%
- Partitioned ASLR: non-measurable
- Random padding: additional 2.8%

Thesis contributions

- New, general defense concept
  - Securing systems by preventing information leaks
- Study of information leaks
  - Providing insights into their causes and prevention
- Discovery of new threats
  - Compilers make mistakes! Uninitialized pointers can be reliably exploited
- General ways to prevent information leaks
  - Three ways to fix root causes and protect certain data
- Novel defense mechanisms
  - Automated and practical design, open sourced implementation

Future work

- Uncovering and fixing classes of logic errors and design flaws
  - No uniform pattern for logic errors or design flaws
  - Empirical analysis and fuzzing
  - Patch history
- Detecting probing (side-channel) attacks
  - Conservative detection + effective defense
  - Transparent detection with hardware features

Conclusions

- Vulnerabilities and insecure designs are common in widely used systems; compilers make mistakes

- This thesis aims to secure widely used systems in an **automated** and **practical** manner

- Preventing information leaks can be a **general** and **practical** solution to defeating both data leaks and control attacks