开源操作系统实践

第二讲 内核静态分析

向勇
清华大学计算机系
xyong@tsinghua.edu.cn

2022年7月

第二讲 内核静态分析

1. Background
2. Static Analysis
3. A Perfect Work

1. Background

1. **Background**
    - **LINT: A Programming Tool**
    - The Problem
    - Compiler
2. Static Analysis
3. A Perfect Work

DOUBLE-CHECKING PROGRAMS: LINT

Checks your program more thoroughly than **cc** does:

```
Utility : lint { fileName }*
```

Lint scans the specified source files and displays any potential errors that it finds.

```
$ lint  reverse.c                       ---> check “reverse.c”.
reverse  defined ( reverse.c(12) ), but never used

$ lint palindrome.c                     ---> check “palindrome.c”.
palindrome  defined ( palindrome.c ( 12 ) ), but never used reverse used ( palindrome.c(14) ), but not defined
```

DOUBLE-CHECKING PROGRAMS: LINT

```
$ lint  main2.c                    ---> check "main2.c".
main2.c(11) : warning: main() returns random value to invocation environment
printf returns value which is always ignored
palindrome used ( main2.c(9) ), but not defined

$ lint  main2.c  reverse.c  palindrome.c ---> check all modules together.
main2.c:
min2.c(11): warning: main() returns random value to invocation environment
reverse.c:
palindrome.c:
Lint pass2:
printf returns value which is always ignored
$ _
```

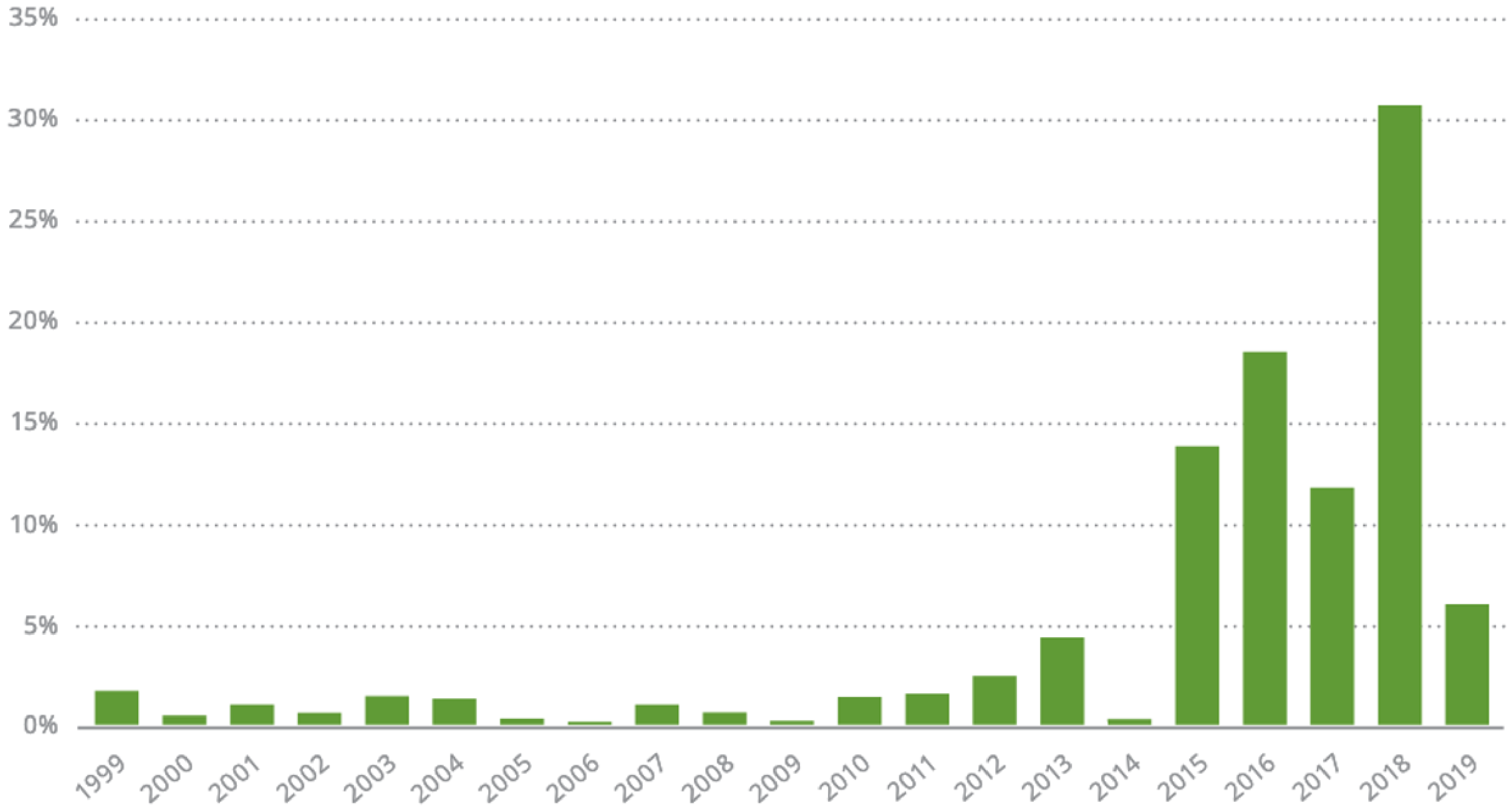
1.2 The Problem

1. Background
    - LINT: A Programming Tool
    - **The Problem**
    - Compiler
2. Static Analysis
3. A Perfect Work

Possible Programming Bugs

- Application
- Programming Language
- User lib & Syscall
- Compiler
- Hardware

Landscape for all discovered CVE in 2019



Vulnerabilities by Category (2016-2020)

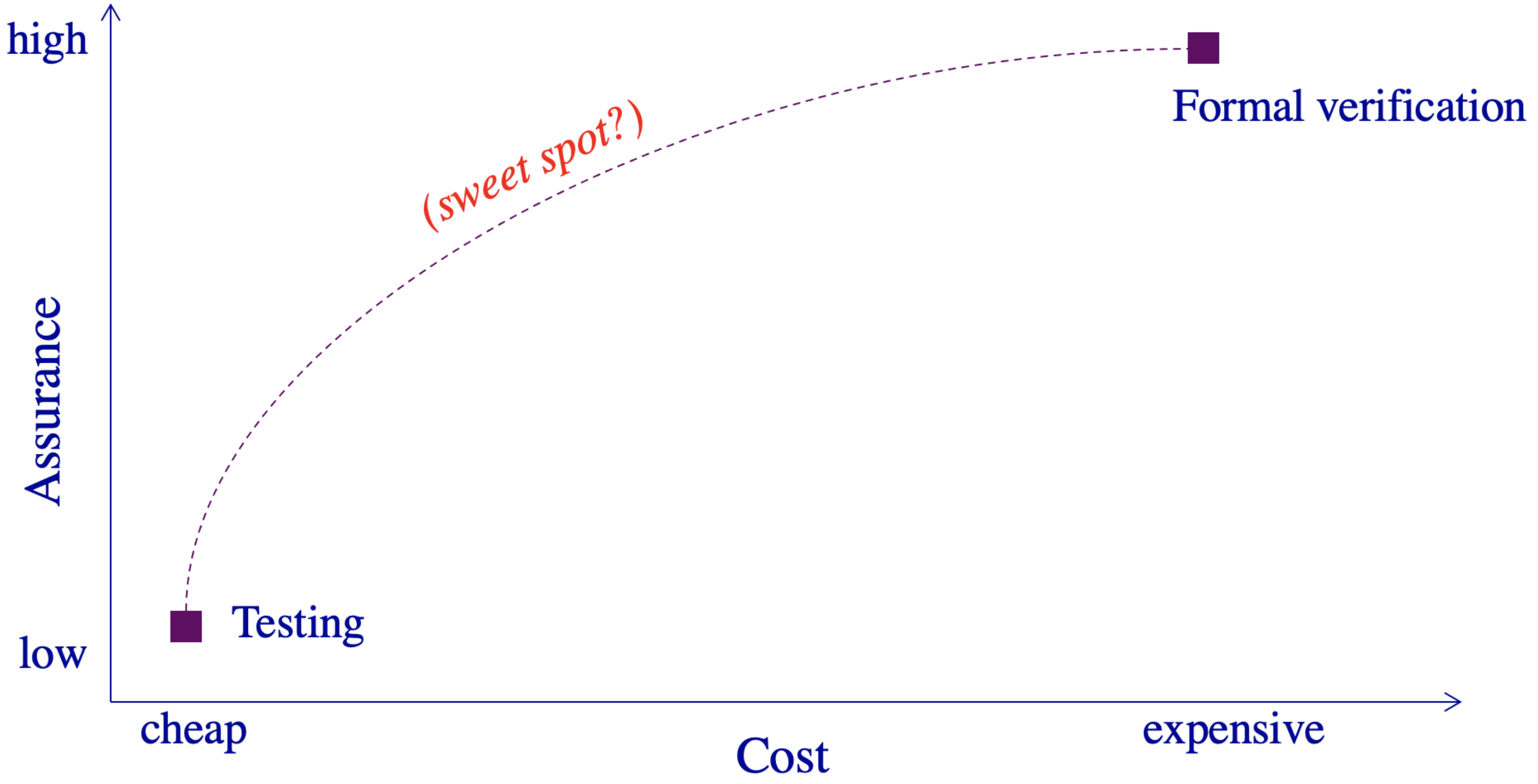
	2020	2019	2018	2017	2016
Remote Coded Execution	345 (27%)	323	292	301	269
Elevation of Privilege	559 (44%)	198	145	90	114
Information Discloure	179 (14%)	177	153	193	102
Denial of Service	46 (4%)	52	29	43	0
Spoofing	104 (8%)	63	20	16	12
Tampering	7 (0.5%)	8	8	1	0
Security Feature Bypass	30 (2.5%)	38	20	41	26

The Problem

- Building secure systems is hard
  - 2/3 of Internet servers have gaping security holes
- The problem is buggy software
  - And a few pitfalls account for many vulnerabilities
- Challenge: Improve programming technology
  - Need way to gain assurance in our software
  - Static analysis can help!

[List of tools for static code analysis]

Existing Paradigms



What Makes Security Hard?

Security is hard because of...

- buffer overruns
- privilege pitfalls
- untrusted data
- ......

1.3 Compiler

1. Background
    - LINT: A Programming Tool
    - The Problem
    - **Compiler**
2. Static Analysis
3. A Perfect Work

What is a compiler?

- A program that translates a program in one language to another language
  - The essential interface between applications & architectures
- Typically lowers the level of abstraction
  - analyzes and reasons about the program & architecture
- We expect the program to be optimized, i.e., better than the original
  - ideally exploiting architectural strengths and hiding weaknesses

Role of compilers

- **Bridge** complexity and evolution in architecture, languages, & applications
- **Help programmers** with correctness, reliability, program understanding
- Compiler **optimizations** can significantly improve performance
    - 1 to 10x on conventional processors
- **Performance stability**: one line change can dramatically alter performance
    - unfortunate, but true

Optimization

What should it do?

1. improve running time, or
2. decrease space requirements
3. decrease power consumption

How does it do it?

Example optimizations

- Division of optimizations
  1. Machine independent
  2. Machine dependent
- Faster code optimizations
  - common subexpression elimination
  - constant folding
  - dead code elimination
  - register allocation
  - scheduling

Analysis

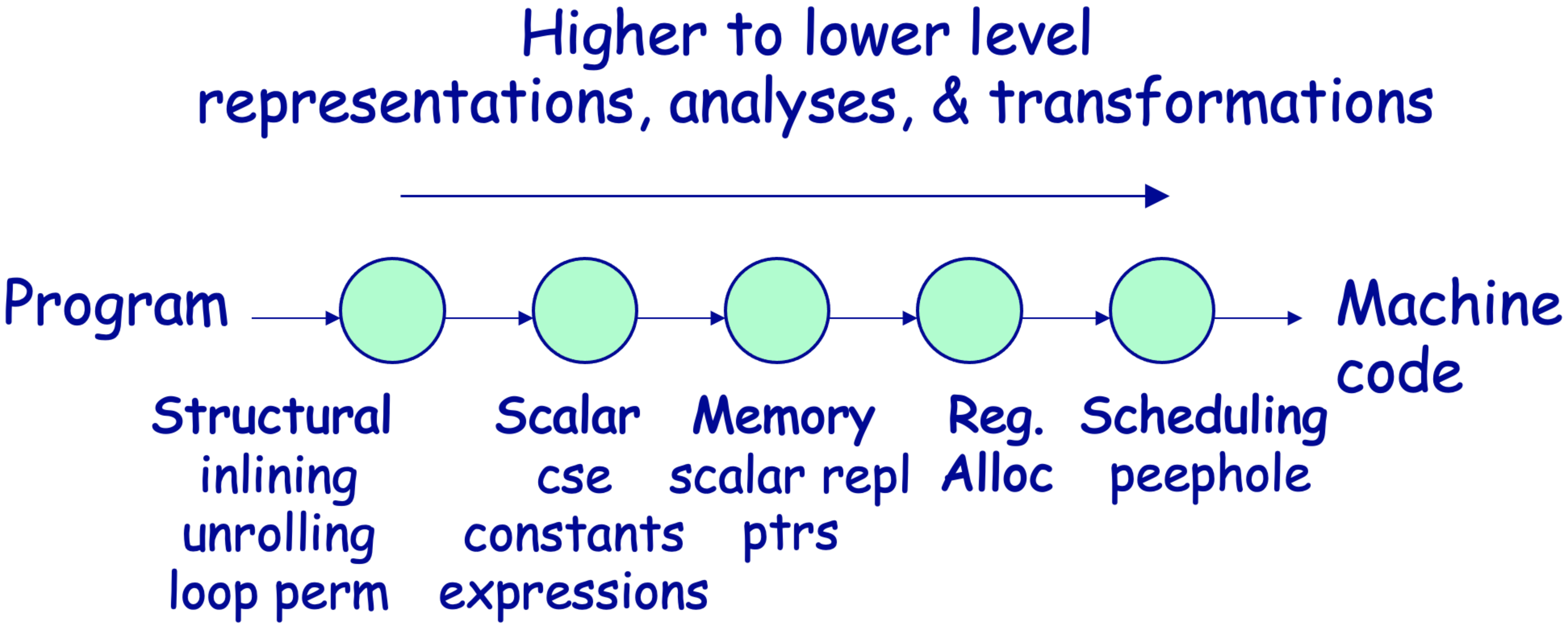
Scope of program analysis

1. within a basic block (local)
2. within a method (global)
3. across methods (interprocedural)

Analysis

1. control flow graph - dominators, loops, etc.
2. dataflow analysis - flow of values
3. static-single-assignment – transform programs such that each variable has a unique definition
4. alias analysis - pointer memory usage
5. dependence analysis - array memory usage

Basic Compiler Structure



2. Static Analysis

1. Background
2. **Static Analysis**
    - **Concept**
    - Buffer Overrun
    - Pitfalls of Privileges
    - Untrusted Data
3. A Perfect Work

2.1 Concept

Static Analysis - Concept

- Examples: compiler optimizations, program verifiers
- Examine program text (no execution)
- Build a **model** of program state
  - An abstraction of the run-time state
- Reason over possible behaviors
  - E.g., “run” the program over the abstract state

Abstract interpretation

- Typically implemented via dataflow analysis
- Each program statement's transfer function indicates how it transforms state
- Example: What is the transfer function for `y = x++;`?

Selecting an abstract domain

$\langle$ x = { 3, 5, 7 }; y = { 9, 11, 13 } $\rangle$

```
y = x++;
```

$\langle$ x = { 4, 6, 8 }; y = { 3, 5, 7 } $\rangle$

$\langle$ x is odd; y is odd $\rangle$

```
y = x++;
```

$\langle$ x is even; y is odd $\rangle$

$\langle$x=3, y=11$\rangle$, $\langle$x=5, y=9$\rangle$, $\langle$x=7, y=13$\rangle$

```
y = x++;
```

$\langle$x=4, y=3$\rangle$, $\langle$x=6, y=5$\rangle$, $\langle$x=8, y=7$\rangle$

$\langle$ x is prime; y is prime $\rangle$

```
y = x++;
```

$\langle$ x is anything; y is prime $\rangle$

$\langle\, x_n = f(a_{n-1},\ldots,z_{n-1});\ y_n = f(a_{n-1},\ldots,z_{n-1})\,\rangle$

```
y = x++;
```

$\langle\, x_{n+1} = x_n+1;\ y_{n+1} = x_n \,\rangle$

Research challenge: Choose good abstractions

- The abstraction determines the **expense** (in time and space)
- The abstraction determines the **accuracy** (what information is lost)
  - Less accurate results are poor for applications that require precision
  - Cannot conclude all true properties in the grammar

Static analysis: Characteristic

- Slow to analyze large models of state, so use abstraction
- Conservative: account for abstracted-away state
- Sound: (weak) properties are guaranteed to be true
  - Some static analyses are not sound

2.2 Buffer Overruns

1. Background
2. Static Analysis
    - Concept
    - **Buffer Overrun**
    - Pitfalls of Privileges
    - Untrusted Data
3. A Perfect Work

What Is a Buffer Overflow

Buffer Overflow

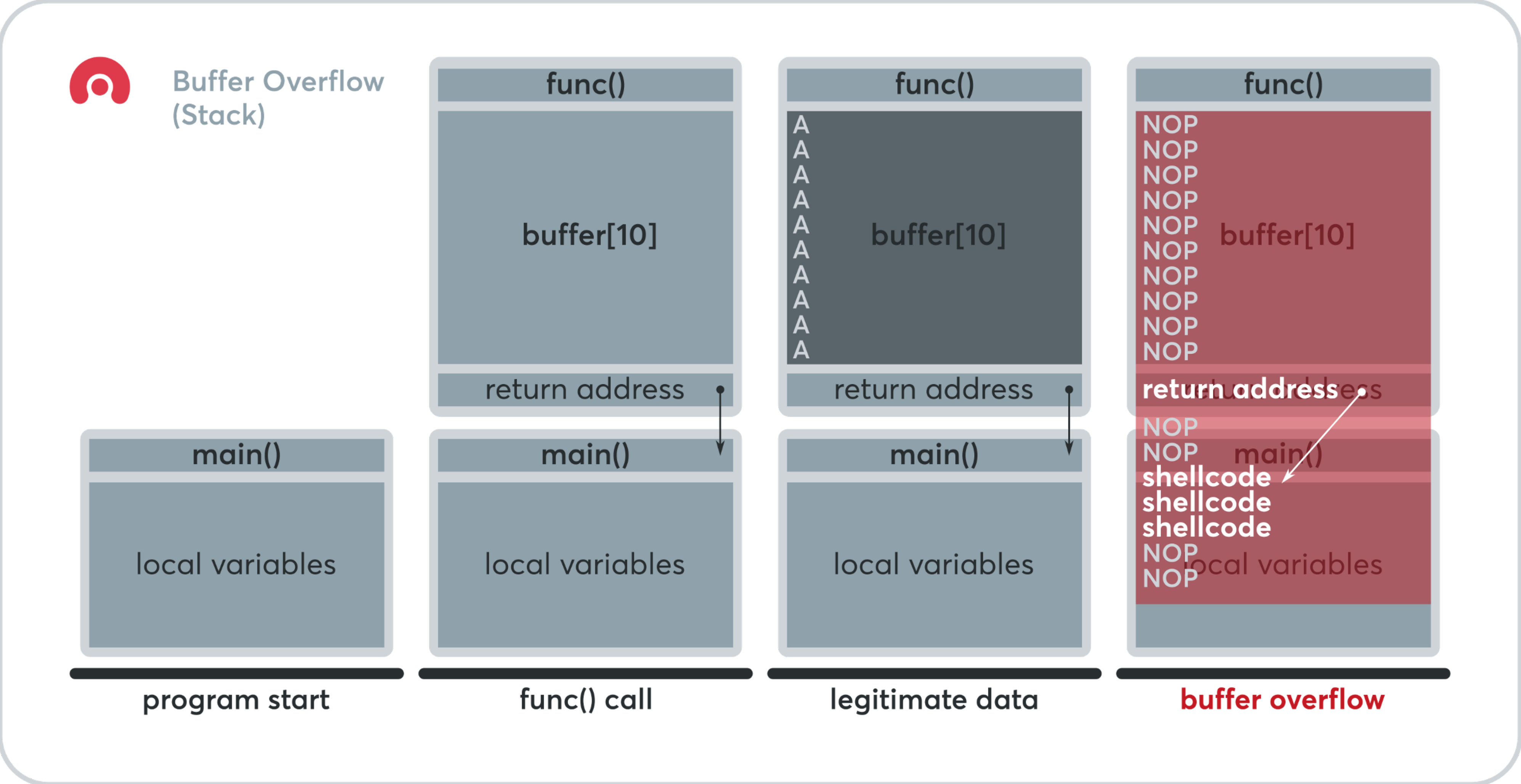
- An example bug

```
main(int argc, char *argv[]) {
  func(argv[1]);
}
void func(char *v) {
  char buffer[10];
  strcpy(buffer, v);
}
```

- Command line：

```
$ vulnprog AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Buffer Overflow



Static Detection of Overruns

- Introduce implicit variables:
  - `alloc(buf) = # bytes allocated for buf`
  - `len(buf) = # bytes stored in buf`
  - Safety condition: `len(buf) ≤ alloc(buf)`

Current Status

- Experimental results
  - Found new bugs in sendmail (30k LOC), others
  - Analysis is fast, but many false alarms (1/kLOC)
- Research challenges
  - Pointer analysis (support strong updates)
  - Integer analysis (infer linear relations, flow-sensitivity)
  - Soundness, scalability, real-world programs

2.3 Pitfalls of Privileges

1. Background
2. Static Analysis
    - Concept
    - Buffer Overrun
    - **Pitfalls of Privileges**
    - Untrusted Data
3. A Perfect Work

Pitfalls of Privileges

- Spot the bug:

```
setuid(0);
rv = bind(...);
if (rv < 0)
   return rv;
seteuid(getuid());
```

Pitfalls of Privileges

- Spot the bug:

enablePriv()

```
setuid(0);
rv = bind(...);
if (rv < 0)
     return rv;
seteuid(getuid());
```

checkPriv()

Bug! Leaks privilege

disablePriv()

A Common Language

- Various interpretations are possible
  - C: enablePriv(p) lasts until next disablePriv(p)
  - Java: ... or until containing stack frame is popped
  - checkPriv(p) throws fatal error if p not enabled

Static Privilege Analysis

- Some problems in privilege analysis:
  - Privilege inference (auditing, bug-finding)
    - Find all privileges reaching a given program point
  - Enforcing privilege-safety (cleanliness of new code)
    - Verify statically that no checkPriv() operation can fail
    - … or that program behaves same under C & Java styles

Future Directions

- Research challenges
  - Experimental studies on real programs
  - Handling data-directed privilege properties
  - Other access control models

2.4 Untrusted Data

1. Background
2. Static Analysis
    - Concept
    - Buffer Overrun
    - Pitfalls of Privileges
    - **Untrusted Data**
3. A Perfect Work

Manipulating Untrusted Data

- Spot the bug:

```
hp = gethostbyaddr(...);
printf(hp->hp_hname);
```

Manipulating Untrusted Data

- Spot the bug:

```
hp = gethostbyaddr(...);
printf(hp->hp_hname);
```

untrusted source of data

Bug! printf() trusts its first argument

Trust Analysis

- Security involves much mental “bookkeeping”
  - Problem: Help programmer keep track of which values can be trusted
- One approach: static taint analysis
  - Extend the C type system
  - Qualified types express annotations: e.g., `tainted char *` is an untrusted string
  - Typechecking enforces safe usage
  - Type inference reduces annotation burden

A Tiny Example

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);

tainted char *s = read_from_network();
printf(s);
```

After Type Inference...

```
void printf(untainted char *, ...);
tainted char * read_from_network(void);
```

an inferred type

```
tainted char *s = read_from_network();
printf(s);
```

Doesn't type-check!
Indicates vulnerability

… where untainted T ≤ tainted T

Current Status

- Experimental results
  - Successful on real programs
    - Able to find many previously-known format string bugs
    - Cost: 10-15 minutes per application
  - Type theory seems useful for security engineering
- Research challenges
  - Richer theory to support real programming idioms
  - More broadly-applicable discipline of good coding
  - Finer-grained notions of trust

Concluding Remarks

- Static analysis can help secure our software
  - Buffer overruns, privilege bugs, format string bugs
  - Hits a sweet spot: cheap and proactive
- Security as a source of interesting problems?
  - Motivations for better pointer, integer analysis
  - New problems: privilege analysis, trust analysis

3. A Perfect Work

- Improving Integer Security for Systems with KINT
  - https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang
- Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior
  - http://sigops.org/sosp/sosp13/program.html

References

1. Lint, a C Program Checker
2. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior
3. Improving Integer Security for Systems with KINT
4. Static Analysis and Software Assurance
5. Static and dynamic analysis: synergy and duality
6. compiler
7. Static/Dynamic Analysis Tools
8. C Programming Tools

谢谢！