ROLoad: Securing Sensitive Operations with Pointee Integrity

Chao Zhang
Tsinghua University

About Me

2004-2008-2013 → 2013-2016 → 2016-present

- Hack for fun
  - 智能软件分析： GeekPwn·AI隐身赛·优胜奖
  - 漏洞挖掘： 腾讯CSS安全探索论坛~专业奖， 300多个CVE漏洞
  - 漏洞利用： 腾讯CSS安全探索论坛~突破奖
  - 漏洞防御： Microsoft BlueHat Prize Contest（特别提名奖）
  - **自动攻防： 美国国防部DARPA CGC**（资格赛防御第一，决赛攻击第二）
  - 人工攻防： DEFCON CTF（2016年第二名，国内破记录）

- Awards/Honors
  - **清华大学学术新人**
  - 求是杰出青年学者
  - 中国科协“青年人才托举工程”
  - 中国区35岁以下科技创新35人（MIT TR35 China
  - **海外某人才计划**
  - 中国计算机学会“青年人才发展计划”

αDiff (ASE'18)
Argot (IJCAI'20)
RAProducer (ISSTA'21)
iDEV (ISSTA'21)

HOTracer (Sec'17)
CollAFL (SP'18)
CollAFL-bin (TDSC'20)
Mopt (Sec'19)
GreyOne (Sec'20)
FANS (Sec'20)
VUL-Dist (ICSE'20)
SaTC (Sec'21)

Revery (CCS'17)
INCITE (INFOCOM'21)
Maze (Sec'21)
CFI-eval (CCS'20)
Vscape (Sec'21)

backdoor (SP'12)
CCFIR (SP'13)
VTint (NDSS'15)
DCG (NDSS'15)
JITScope (INFOCOM'15)
VTrust (NDSS'16)
DRAMD (INFOCOM'20)
POP&PUSH (NDSS'21)
ROLoad (DAC'21)
ZKCPlus (CCS'21)

Hacking Practice

蓝莲花战队（教练）

- 2013 大陆首次打入决赛；
- 2014 第五名；
- 2015 第五名；
- **2016 第二名；(人机大战)**
- 2017 第五名；
- 2018 第六名；
- 2019 第三名

DEFCON CTF 战绩

历史前几名

- 2013： ppp, men in black hats, raon_ASRT
- 2014： ppp, hitcon, dragonsector,
- 2015： defkor, ppp, 0daysober, hitcon
- 2016： ppp, b1o0p, defkor, hitcon
- 2017： ppp, hitcon, a*0*e, defkor, tea deliverers
- 2018： defkoroot, ppp, hitcon, a*0*e, sauercloud, td
- 2019: ppp, hitcon, tea-deliverers

Hacking Research

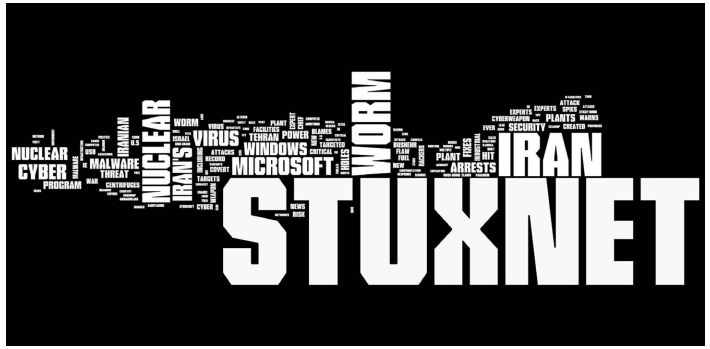
DARPA Cyber Grand Challenge

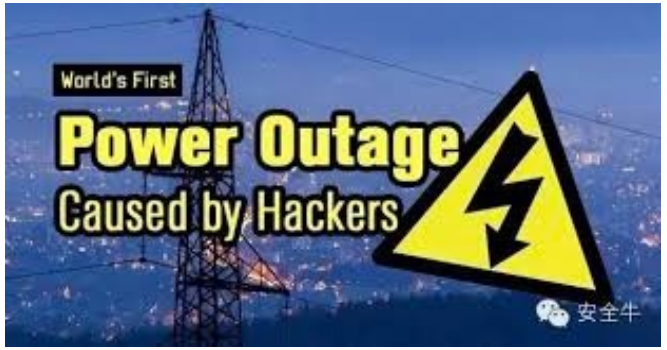
机器自动攻防超级挑战赛

（CodeJitsu队长，资格赛防御#1，决赛攻击#2）

The Problem: Vulnerability

- Software vulnerabilities are major root causes of security incidents.

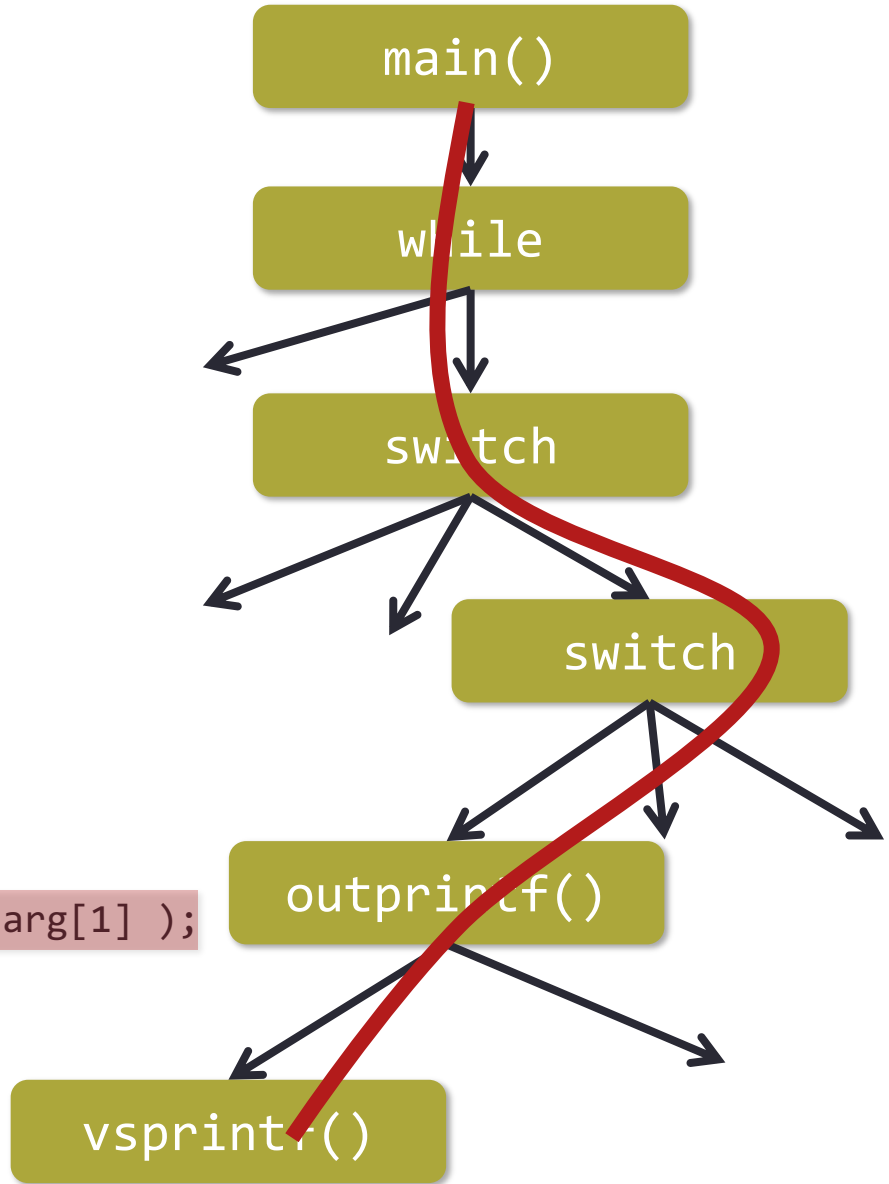




Sample Vulnerability : CVE-2009-4270

```
int outprintf( const char *fmt, … )
{
  int count; char buf[1024]; va_list args;
  va_start( args, fmt );
  count = vsprintf( buf, fmt, args );
  outwrite( buf, count ); // print out
}

int main( int argc, char* argv[] )
{
  const char *arg;
  while( (arg = *argv++) != 0 ) {
    switch ( arg[0] ) {
    case '-': {
      switch ( arg[1] ) {
      case 0:
      …
      default:
        outprintf( "unknown switch %s\n", arg[1] );
      }
    }
    default: …
    }
  …
```



Conditions

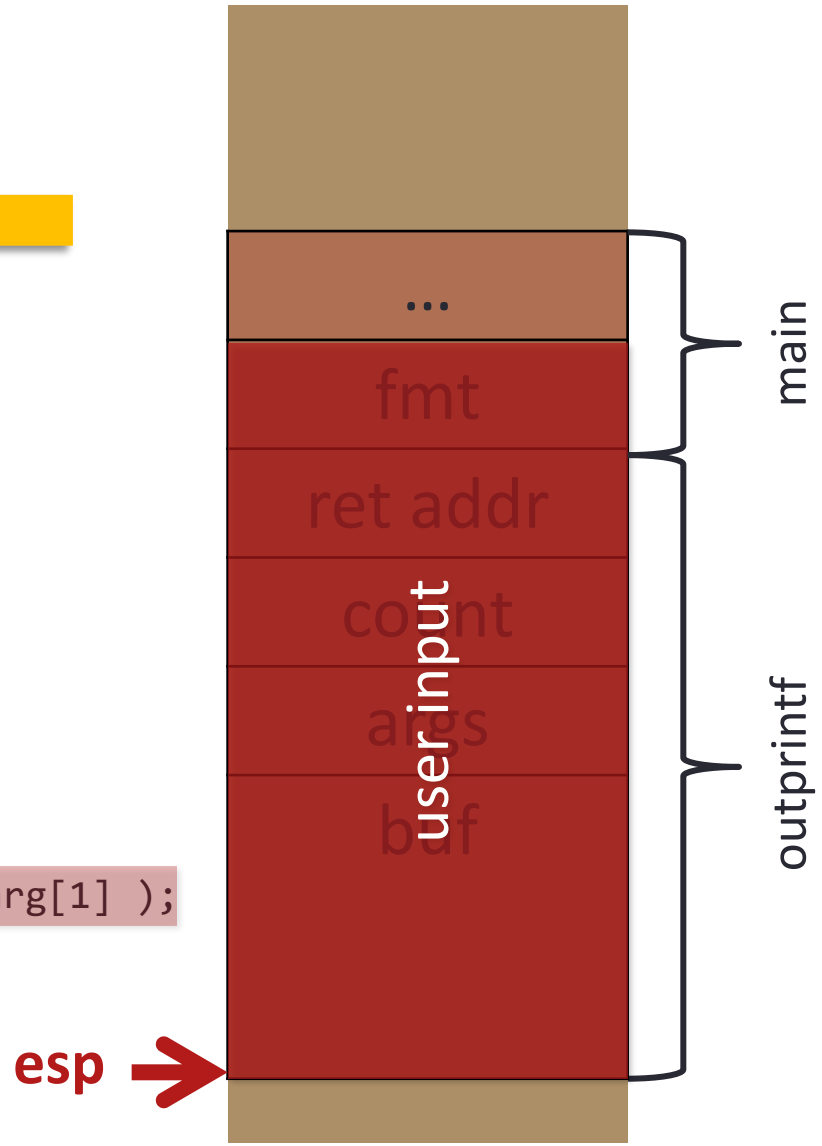
- Execute the specific path
- Input length is large enough
  - The buffer overflow vulnerability constraint

Exploit Vulnerability : CVE-2009-4270

```
int outprintf( const char *fmt, … )
{
  int count; char buf[1024]; va_list args;
  va_start( args, fmt );
  count = vsprintf( buf, fmt, args );
  outwrite( buf, count ); // print out
}
```

Function returns

```
int main( int argc, char* argv[] )
{
  const char *arg;
  while( (arg = *argv++) != 0 ) {
    switch ( arg[0] ) {
    case '-': {
      switch ( arg[1] ) {
      case 0:
      …
      default:
        outprintf( "unknown switch %s\n", arg[1] );
      }
    }
    default: …
    }
  …
```



Hijack points (sensitive ops)

- Function return sites
- Indirect function call sites
- Sensitive API invocations
  - e.g., system()
- Critical data reference
  - E.g., uid
- …

Consequences

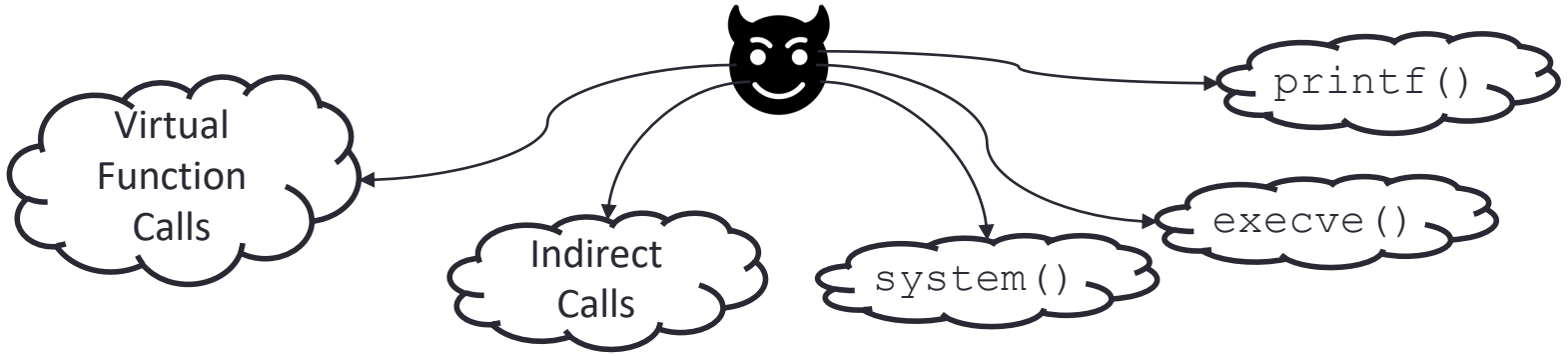
- Control flow hijacking
  - Execute malicious code
- Information leakage
- Privilege escalation
- Denial of service
- …

How to mitigate?

ROLoad: Securing Sensitive Operations with Pointee Integrity

Motivation

- Attackers often exploit memory corruption vulnerabilities by corrupting operands of **sensitive operations**.



- It's necessary to protect sensitive operations to mitigate the risk.

Existing Defense Mechanisms

Very strong security guarantees, but high overheads

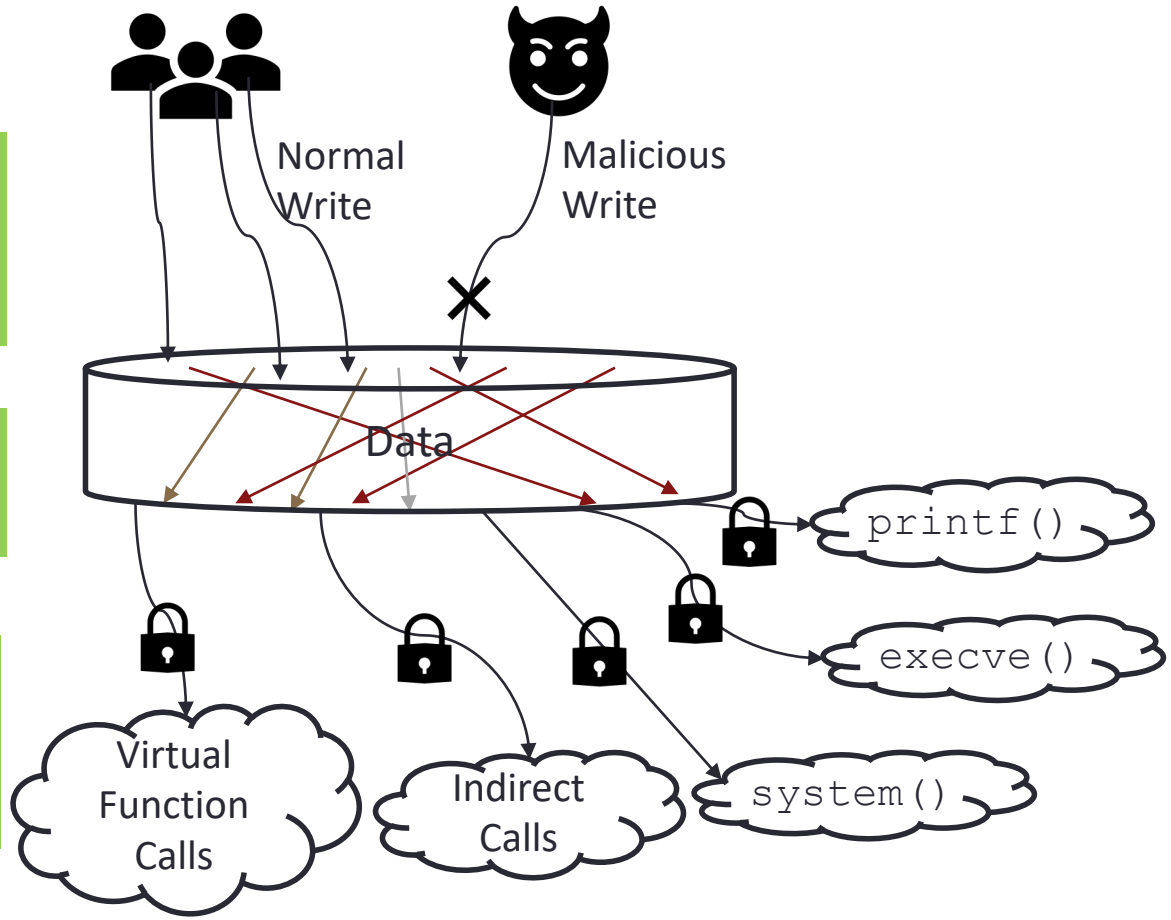
Prevention at Sources (memory safety, ASan, ARM MTE)

Hard to clearly specify the boundary

Data-Flow Isolation (IMIX, HDFI, Intel MPK)

Specific issues: key management problems, or only coarse-grained CFI

Detection at Sinks (ARM PA, Intel CET, ARM BTI)



Intuition: ROLoad

Goal: Detection at sinks, lightweight, strong security guarantee

Q1: How to differentiate good/bad data?

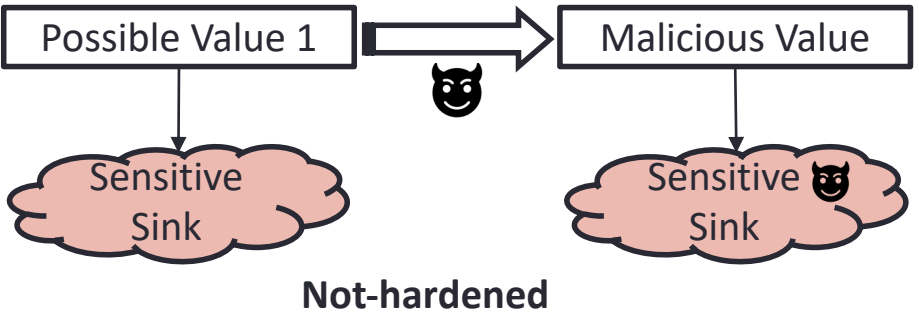
- Good data: in read-only memory
- Bad data: in writable memory

Q2: How to stop bad data from being used at sinks?

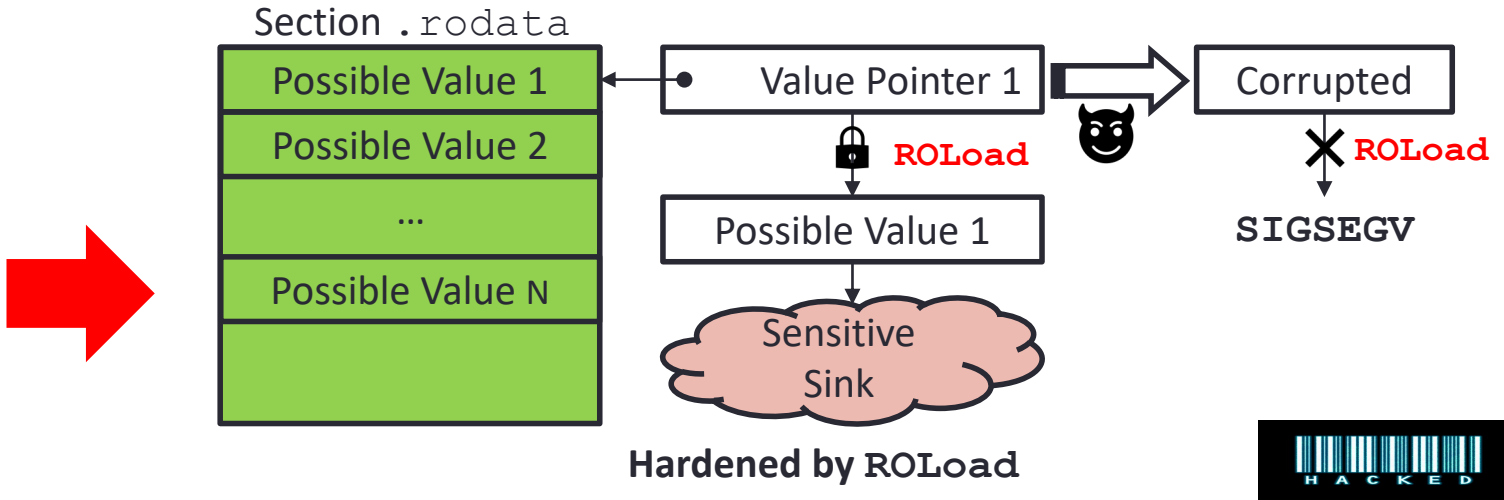
- A special memory load instruction: only loads data from read-only mem
- Sinks only use values loaded by such instructions

Q3: How to ensure legitimate data usable at sinks?

- Place legitimate values at read-only memory first (at compile time)



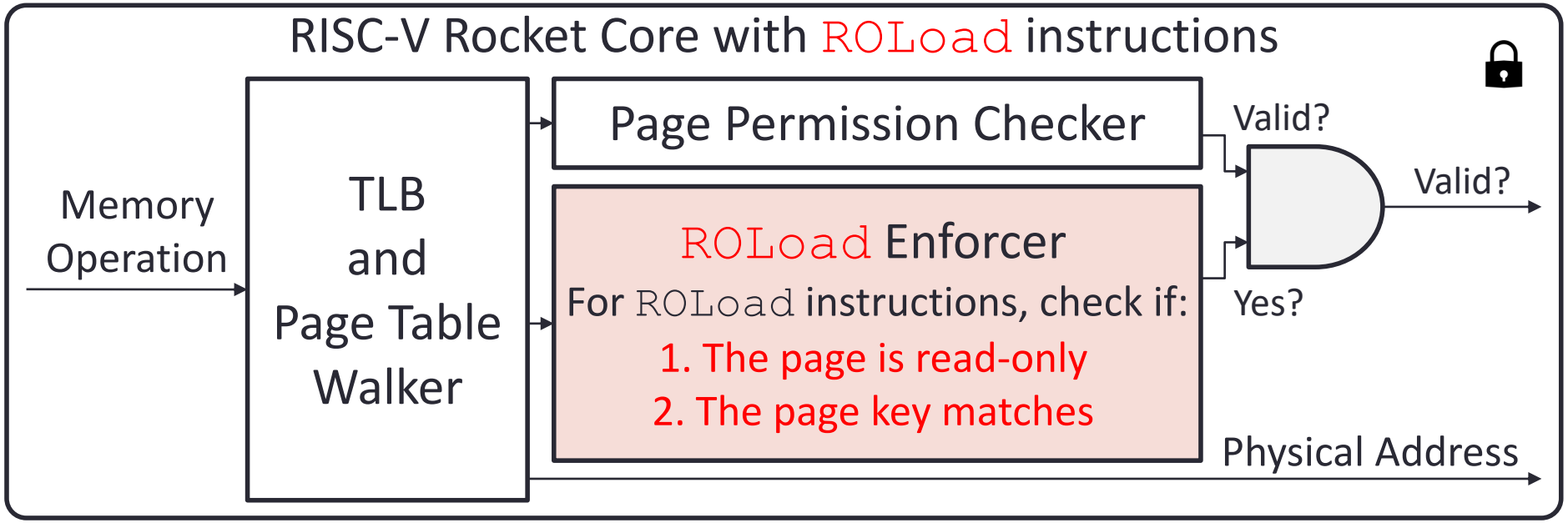
Not-hardened



Hardened by ROLoad

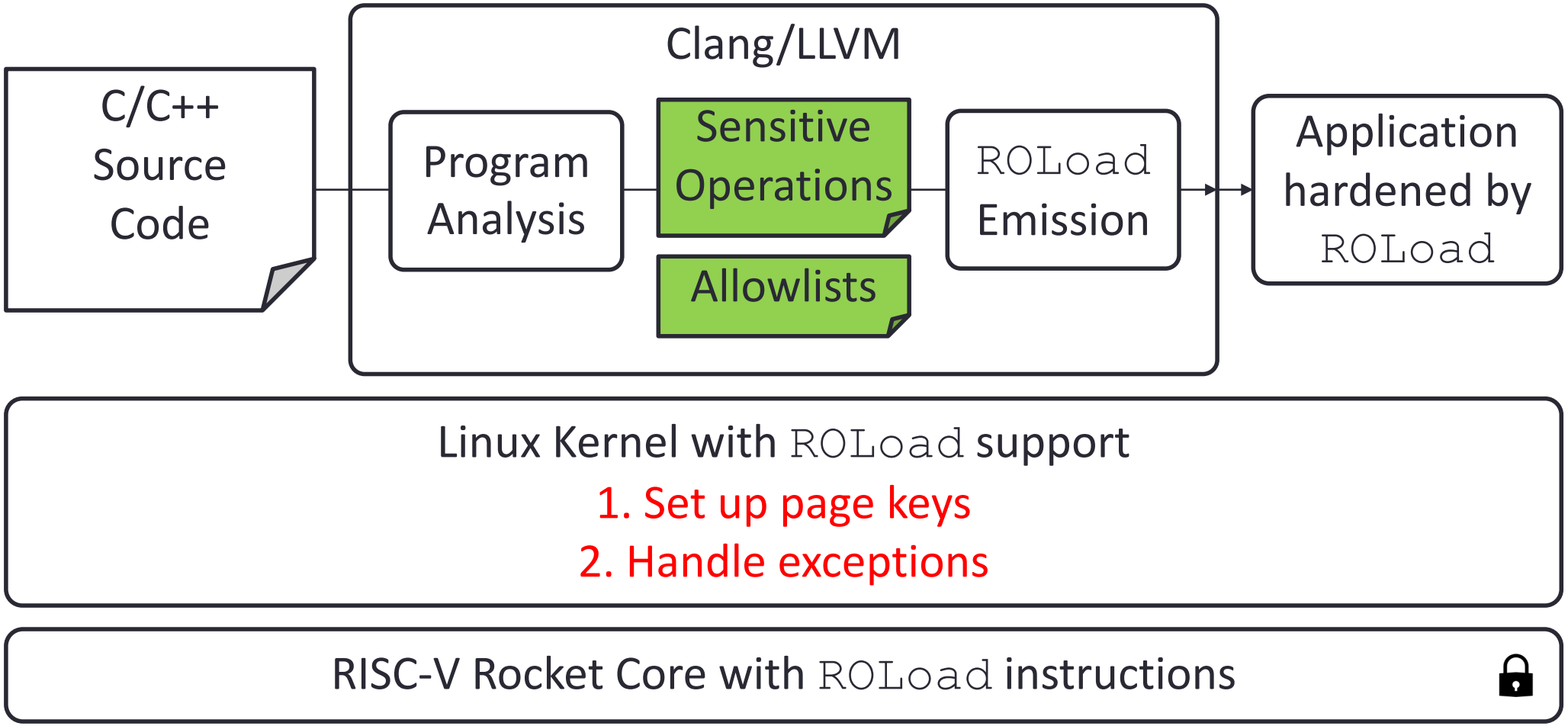
Design and Implementation

- The processor core, operating system kernel, and compiler need to be slightly modified.



Design and Implementation

- The processor core, operating system kernel, and compiler need to be slightly modified.



Application 1: Virtual Function Call Protection

- Replace virtual-table-load instructions with `ROLoad` instructions

```
1  struct foo {
2    virtual void func()
3    {}
4  };
5  struct bar : foo {
6    void func() {}
7  };
8  foo *f, *b;
9  // Init f and b...
10 f->func();
11 b->func();
```

Listing 1: C++ code of calling virtual functions of two C++ objects.

```
1   42 65 ld     a0, 16(sp)
2   0c 61 ld     a1, 0(a0)
3  -8c 61 ld     a1, 0(a1)
4  +8c 9d ld.ro  a1, (a1), 7
5   82 95 jalr   a1
6
7   22 65 ld     a0, 8(sp)
8   0c 61 ld     a1, 0(a0)
9  -8c 61 ld     a1, 0(a1)
10 +8c 9d ld.ro  a1, (a1), 7
11  82 95 jalr   a1
```

Listing 2: Assembly code of calling two virtual functions, hardened by `ROLoad`.

Application 2: Type-Based Forward-Edge CFI

- Store targets of indirect transfers into read-only sections
- Replace code pointers with pointers to them
- Retrieve them back using `ROLoad` just before indirect transfers

```
1  typedef void (*func1_t)(...);
2  typedef int (*func2_t)(...);
3  func1_t func1;
4  func2_t func2;
5  // ...
6  func1 = foo;
7  func2 = bar;
8  // ...
9  func1();
10 func2();
```

Listing 1: Pseudo code of two indirect call examples.

```
1  -lui  a0, 0x11
2  -addi a0, a0, 604    # foo
3  +lui  a0, 0x67       # mem page addr
4  +addi a0, a0, 8      # gfpt_foo
5   sd   a0, -1608(gp)  # func1
6   ...
7  -lui  a0, 0x11
8  -addi a0, a0, 616    # bar
9  +lui  a0, 0x68       # gfpt_bar
10  sd   a0, -1600(gp)  # func2
```

Listing 2: Assembly code of initializing 2 function pointers for `ROLoad`.

```
1   ld    a0, -1608(gp) # func1
2  +ld.ro a0, (a0), 111
3   jalr  a0
4   ld    a0, -1600(gp) # func2
5  +ld.ro a0, (a0), 222
6   jalr  a0
7  +.section .rodata.key.111
8  +gfpt_foo: .quad foo
9  +.section .rodata.key.222
10 +gfpt_bar: .quad bar
```

Listing 3: Assembly code of two indirect calls, hardened by `ROLoad`.

Other Application Scenarios

- Backward control-flow transfers (return instructions)
- Kernel protections (e.g. device descriptors, operation structures)
- …

Evaluation – Prototype System

- <3.32% FPGA resources overheads (core without peripherals)
- The maximum frequency is almost not affected
- SPEC CINT2006 benchmark suite

Components	Configurations
ISA Extensions	RV64IMAC with M, S, and U modes
Caches	32KiB 8-way L1I$, 32KiB 8-way L1D$
TLBs	32-entry I-TLB, 32-entry D-TLB (default)
Peripherals	Xilinx MIG for a 4GiB DDR3 SO-DIMM Xilinx AXI Ethernet Subsystem, 64KiB boot ROM

Evaluation – Applications

- Virtual Function Call Protection
  - 3 benchmarks written in C++ are evaluated
  - 0.31% execution time overheads and 0.035% memory overheads

- Type-Based Forward-Edge CFI
  - ~0% execution time overheads and 0.086% memory overheads

✓`ROLoad` applications are practical to deploy

Takeaways

- A lightweight hardware-software co-design based solution, namely `ROLoad`, securing sensitive operations
- A prototype system based on an FPGA
- Outperforms state-of-the-art solutions
  - Provides comparable defenses as ARM PAC and MTE
  - Costs much lower hardware and runtime overheads
  - Also suitable for IoT, PK systems…

Thanks!

Q&A

https://netsec.ccert.edu.cn/chaoz