Missing the Point(er): On the Effectiveness of Code Pointer Integrity

陈渤、邓毓峰、丁峰

2017年11月9日

Missing the Point(er):
On the Effectiveness of Code Pointer Integrity[1]

Isaac Evans[*], Sam Fingeret[†], Julián González[†], Ulziibayar Otgonbaatar[†], Tiffany Tang[†], Howard Shrobe[†], Stelios Sidiroglou-Douskos[†], Martin Rinard[†], Hamed Okhravi[*]
[†]MIT CSAIL, Cambridge, MA
Email: {samfin, jugonz97, ulziibay, fable, hes, stelios, rinard}@csail.mit.edu
[*]MIT Lincoln Laboratory, Lexington, MA
Email: {isaac.evans, hamed.okhravi}@ll.mit.edu

Contents

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

1

PART ONE

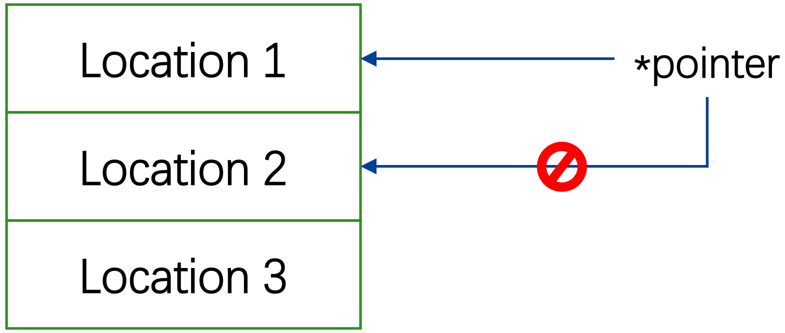
Background

Problem：

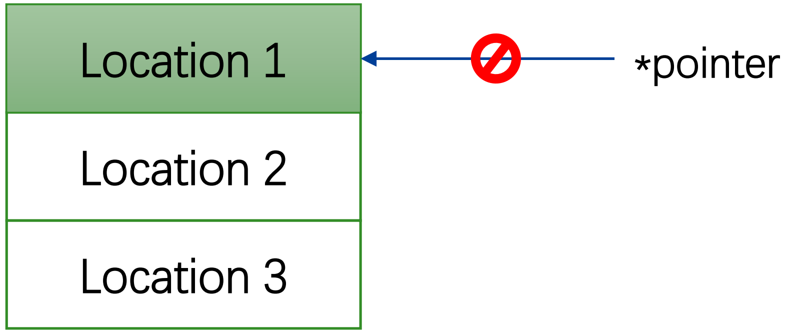
- Unmanaged languages (C/C++)
- Memory corruption and security vulnerabilities

Background:

Complete Memory Safety:



Spatial Pointer Safety



Temporal Pointer Safety

Background:

Characteristic of CPI:

- static analysis

- protects the sensitive pointers

- stores the metadata for checking the validity of code pointers in its safe region

2

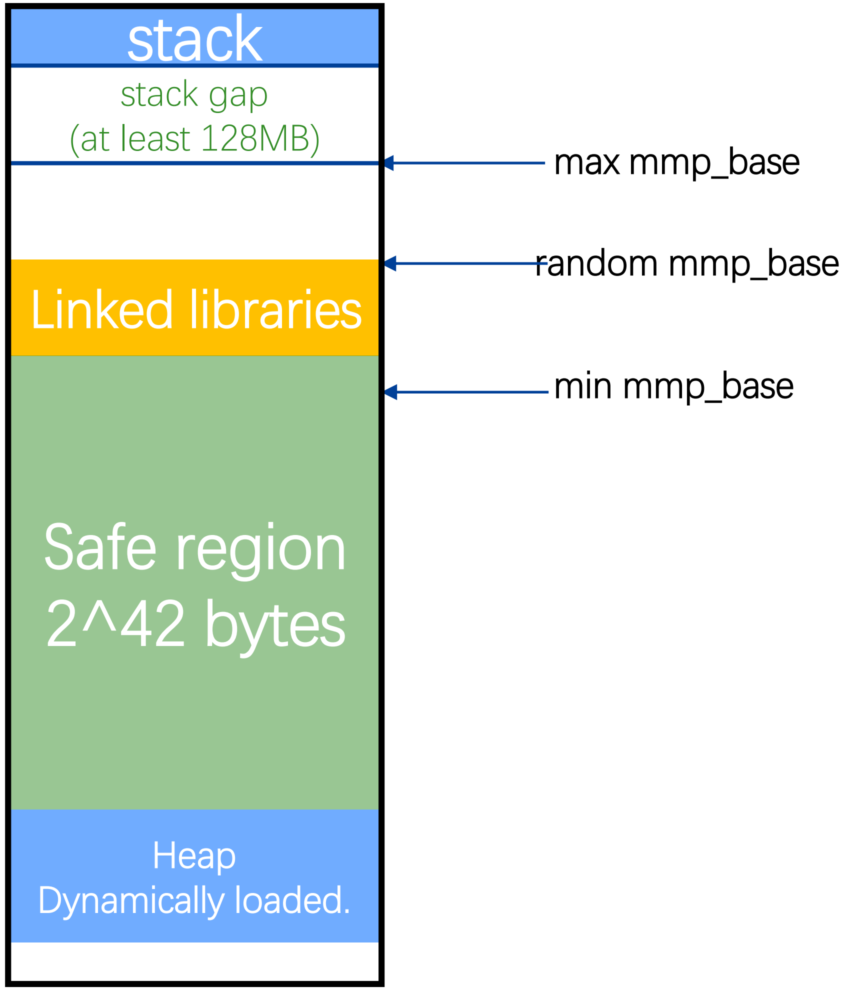
PART TWO

Attack overview

Threat model:

Assumption：

- existing a vulnerability to control the stack
- attacker cannot modify code in memory
- the operating system supports ASLR
- CPI is properly configured and correctly implemented



Safe Region Memory

Attack:

Two design weaknesses in CPI:

- Information hiding (x86-64 and ARM)
- Protection code pointers, not data pointers

Attack:

- Launch Timing Side-channel Attack

control a data pointer and point to a return address on the stack

CPI Underlying Assumptions:

- ~~Assumption~~ 1: If there is no pointer to a region in memory, its content cannot be leaked or corrupted.
- Assumption 2: Leaking large parts of memory requires a prohibitive number of program crashes.

Leak information
- directly
- indirectly, using side-channel attacks
  - timing analysis
  - fault analysis

Inf Leak without Memory Disclosure:

Attacker

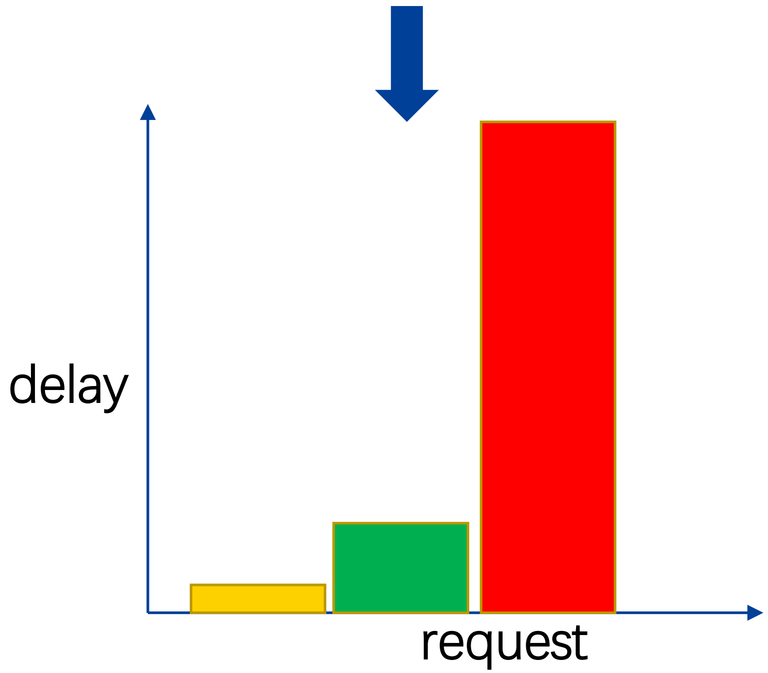
Corrupt *n* to desired memory address

request

Server (NginX)

```
i = 0;
while (i < *n){
    process();
i++;}
```

response



n

Memory values
0x00
0x03
0xFF
…
…

CPI Underlying Assumptions:

- ~~Assumption~~ 1: If there is no pointer to a region in memory, its content cannot be leaked or corrupted.
- ~~Assumption~~ 2: Leaking large parts of memory requires a prohibitive number of program crashes.

Crashing attack	6s
Non-crashing attack	98h

3

PART THREE

Attack methodology

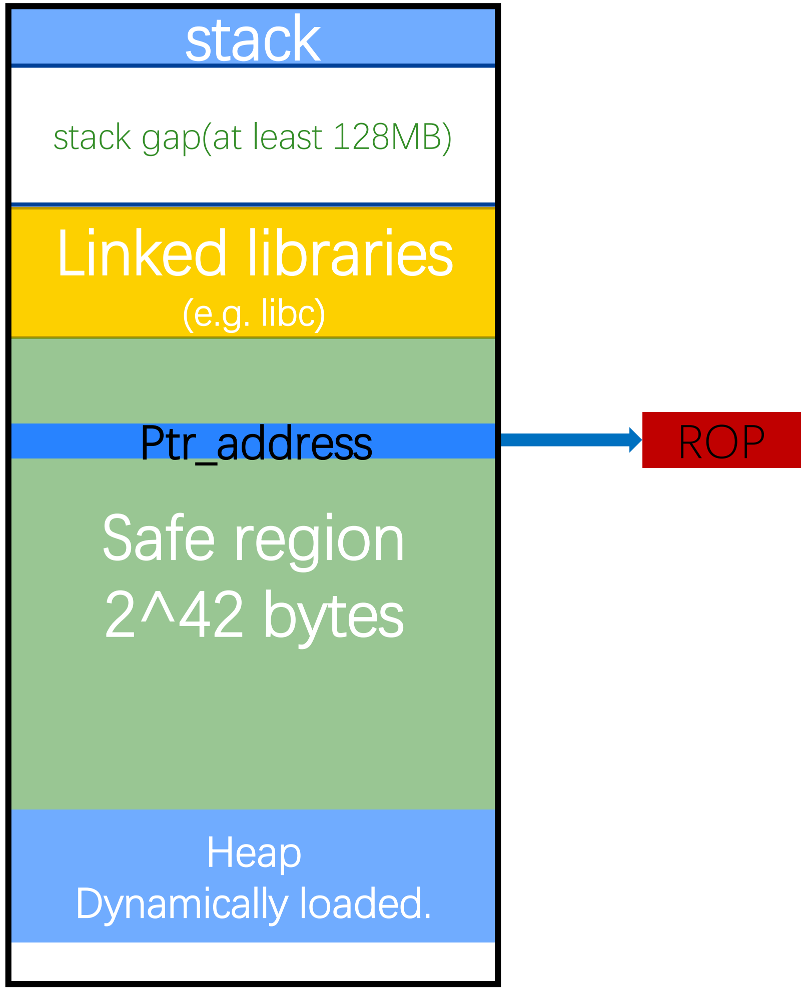
Attack Methodology:

The attack performs the following steps:

1) Data Collection

2) Locate safe region

3) Attack safe region



Safe Region Memory

1) Data Collection

Attacker

Corrupt n to Desired memory address

request

Server

```
While(i<*n)
{
        Process();
}
```

response

$$byte = c \sum_{i=1}^{s} (\boxed{d_i} - baseline) \qquad (1)$$

Memory values
0x00
0x05
0xFF
0x??

n

1) Data Collection

Attacker

Corrupt n to
Desired memory
address

request

Server

```
While(i<*n)
{
        Process();
}
```

response

$$byte = c\sum_{i=1}^{s}(d_i - \boxed{baseline}) \qquad (1)$$

$$baseline = \frac{\sum d_i}{s}$$

n →

Memory values
0x00
0x05
0xFF
0x??

1) Data Collection

Attacker

Corrupt n to Desired memory address

request

Server

```
While(i<*n)
{
        Process();
}
```

response

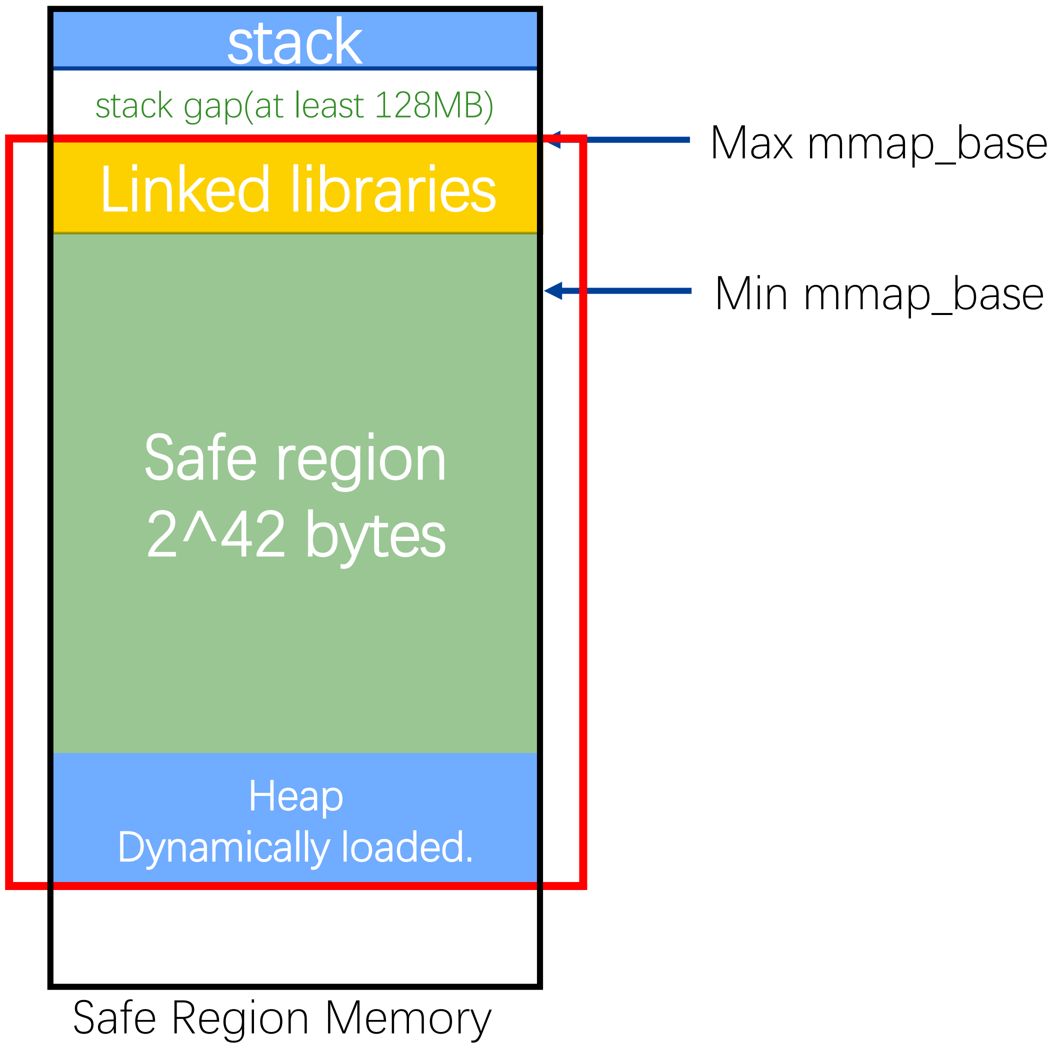
$$byte = \boxed{c}\sum_{i=1}^{s}(d_i - baseline) \qquad (1)$$

$$c = \frac{255}{\sum_{i=1}^{s}(d_i) - s * baseline}$$

Memory values
0x00
0x05
0xFF
0x??

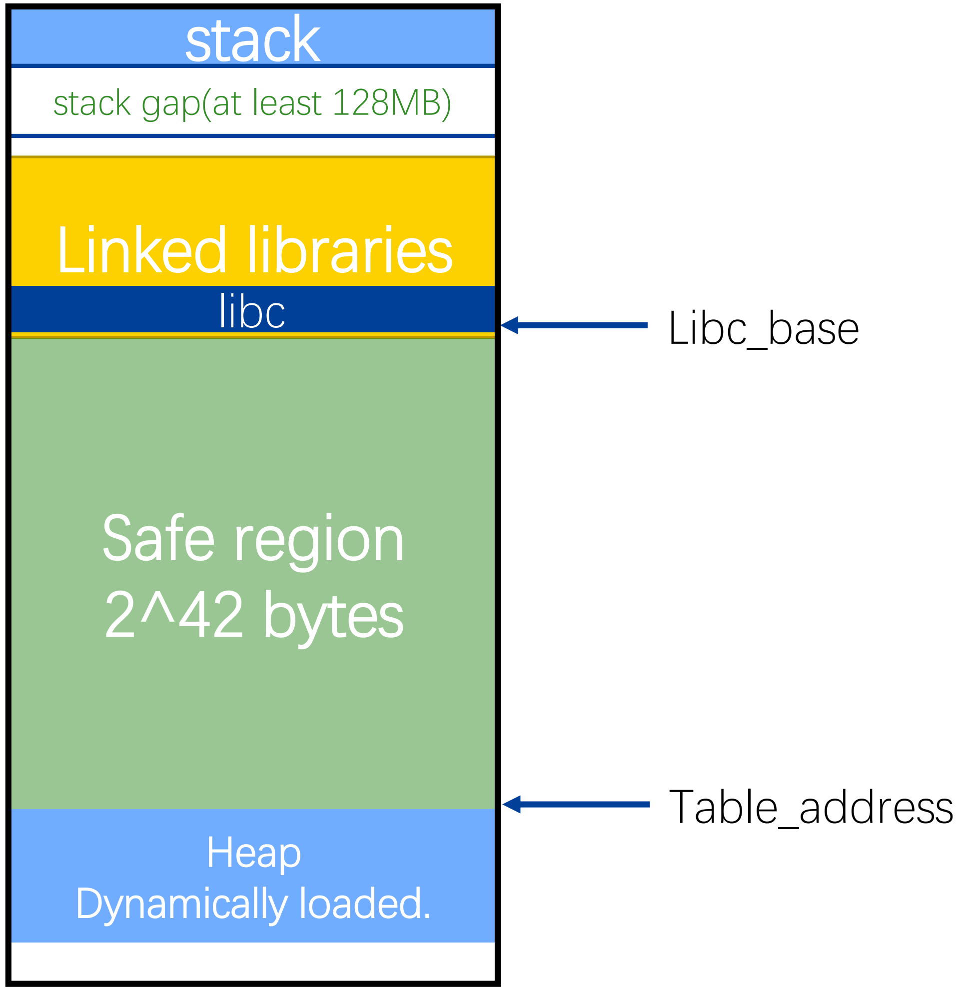
n

2) Locate Safe Region



$$min\_mmap\_base = max\_mmap\_base - aslr\_entropy * page\_size$$

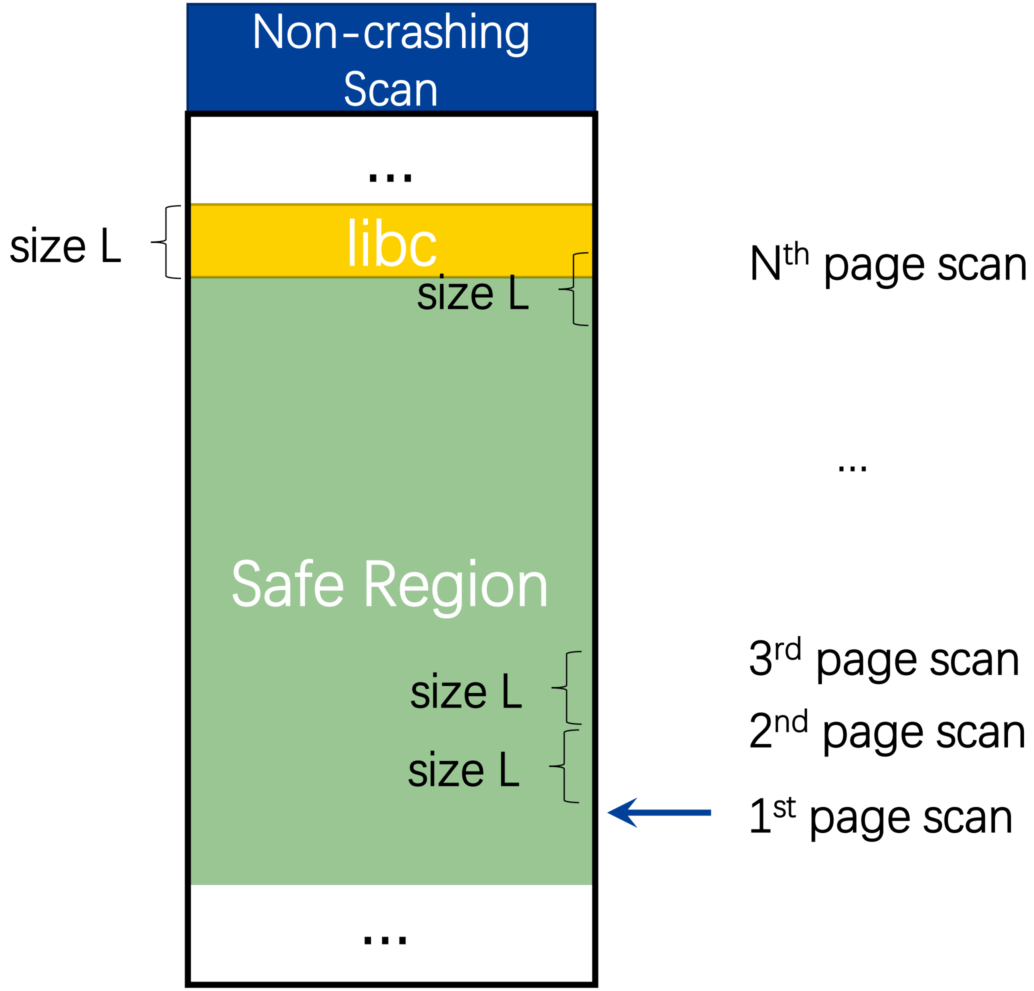
2) Locate Safe Region



Safe Region Memory

$$table\_address = libc\_base - 2^{42}$$

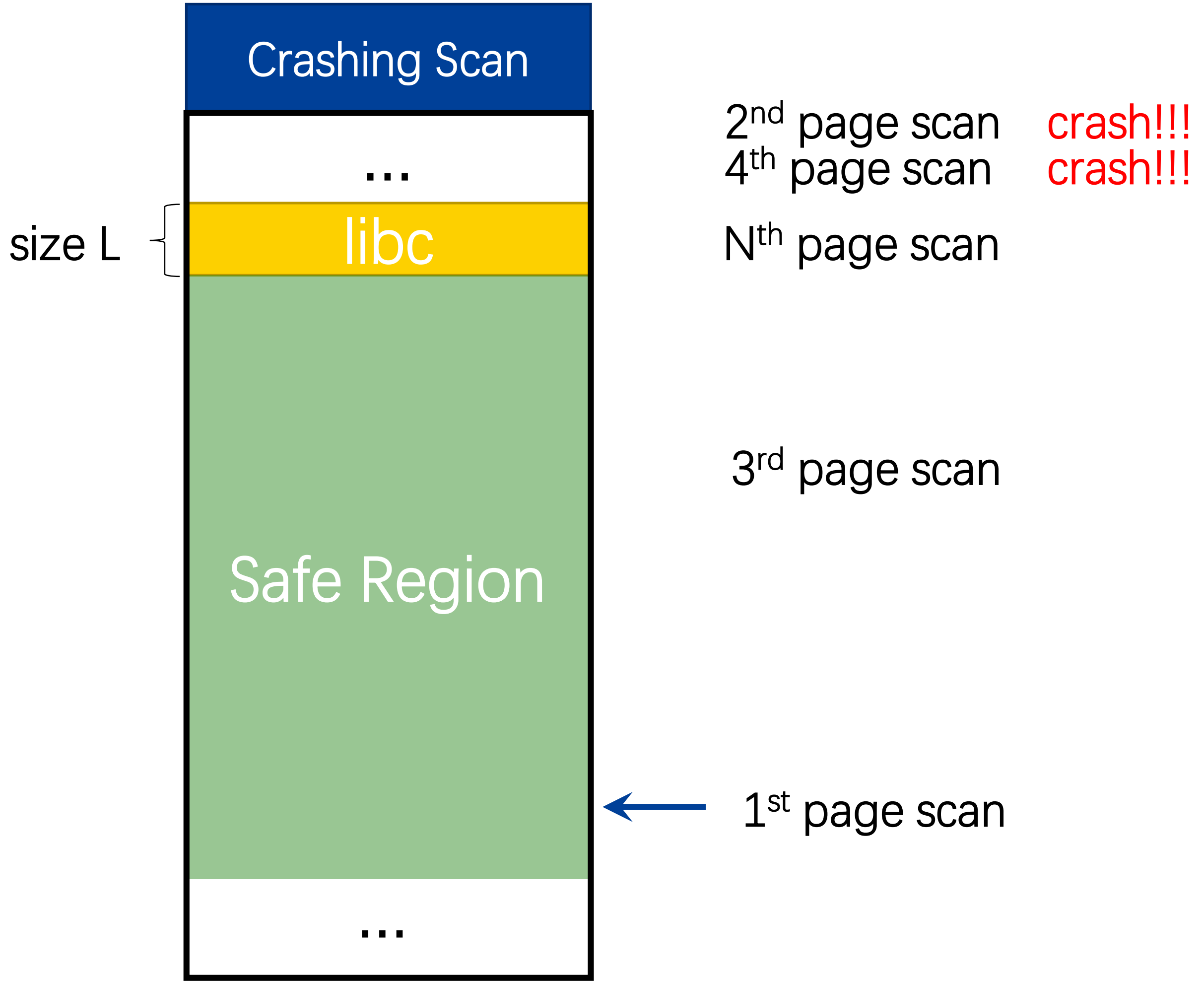
2) Locate Safe Region

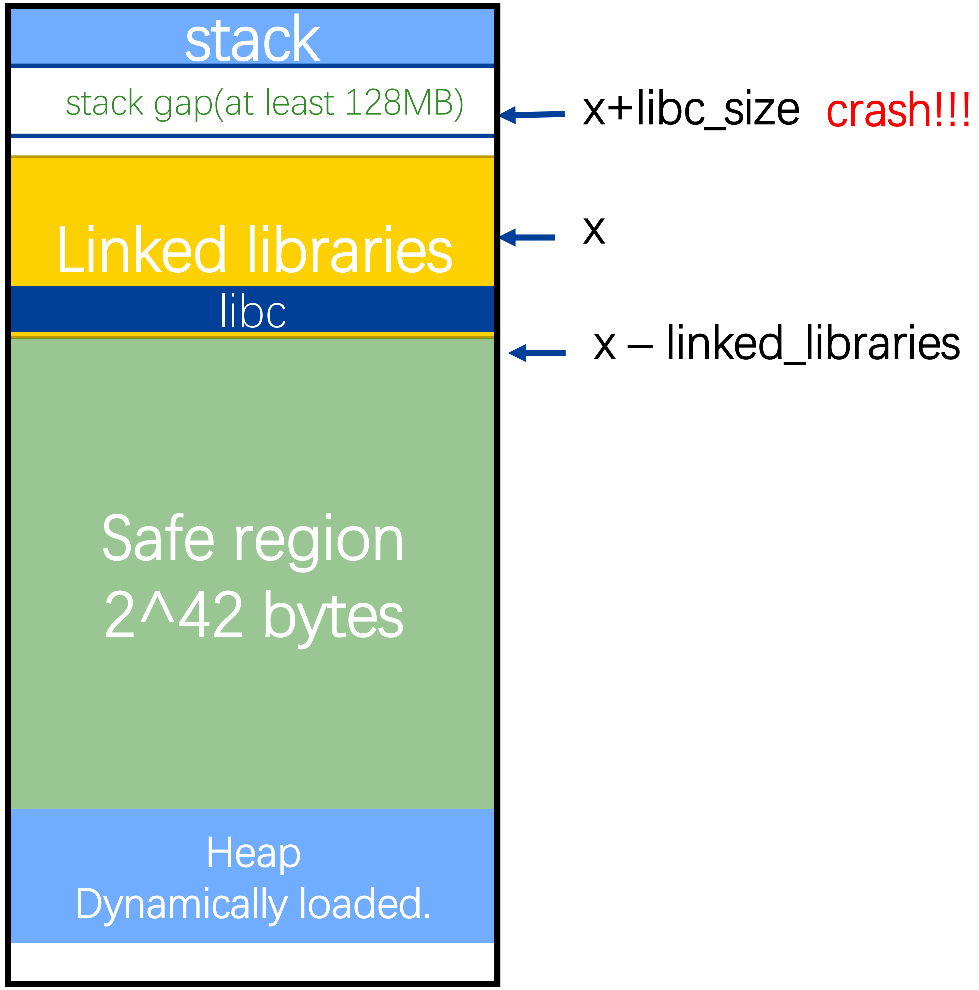


The number of scans in the worst case:

$$(aslr\_entropy * page\_size)/libc\_size$$

2) Locate Safe Region





2) Locate Safe Region

- More generally, T crashes are allowed for our scanning, We use dynamic programming to find the optimum scanning strategy for a given T.

$$f(i,j) = f(i,j-1) + f(i-1,j-1) + 1$$

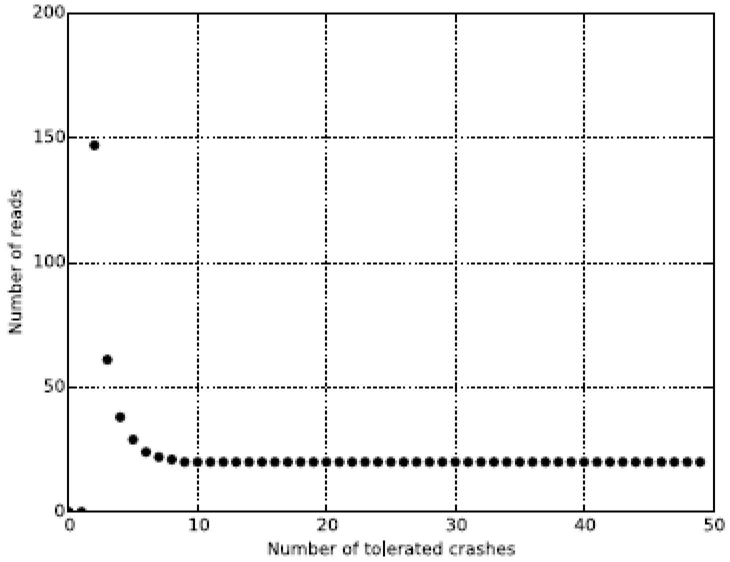


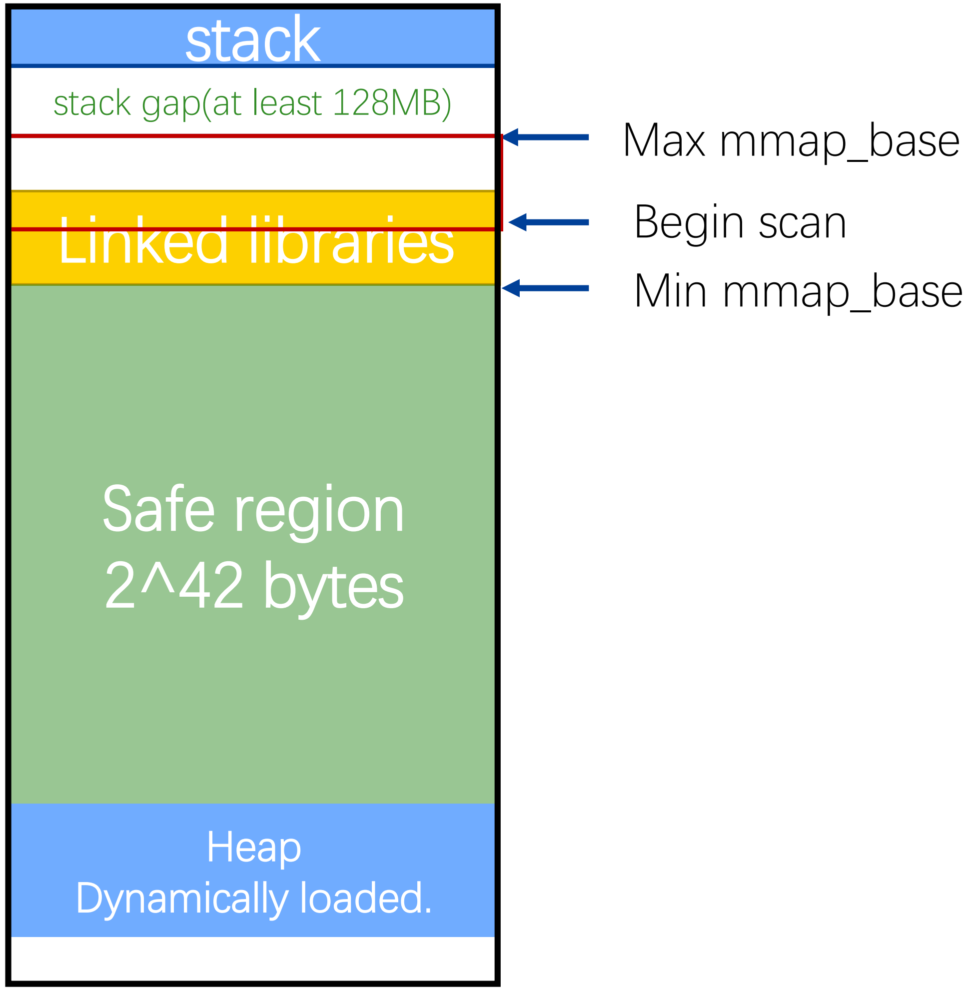
Fig. 3. Tolerated Number of Crashes

- Let $f(i, j)$ be the maximum amount of memory an optimum scanning strategy can cover, incurring up to i crashes, and performing j page reads. Note that to cause a crash, you need to perform a read.

2) Locate Safe Region

- We can still obtain a significant improvement even if the application does rerandomize its address space when it restarts after a crash. Suppose that we can tolerate *T* crashes on average. We will begin our scan at address:

$$max\_mmap\_base - \frac{1}{T+1}(aslr\_entropy * page\_size)$$

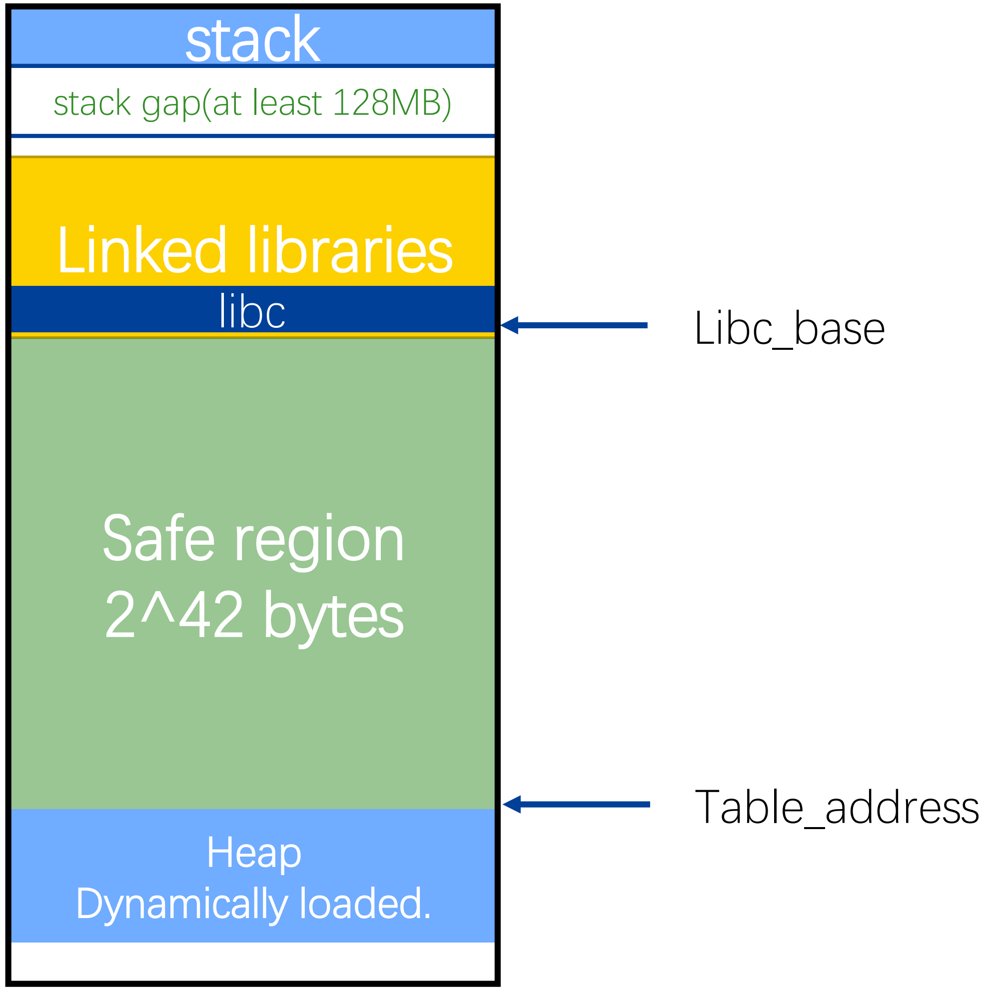


Safe Region Memory

2) Locate Safe Region

Once the base address of mmap is discovered using the timing side channel, the address of the safe region table can be computed as follows:

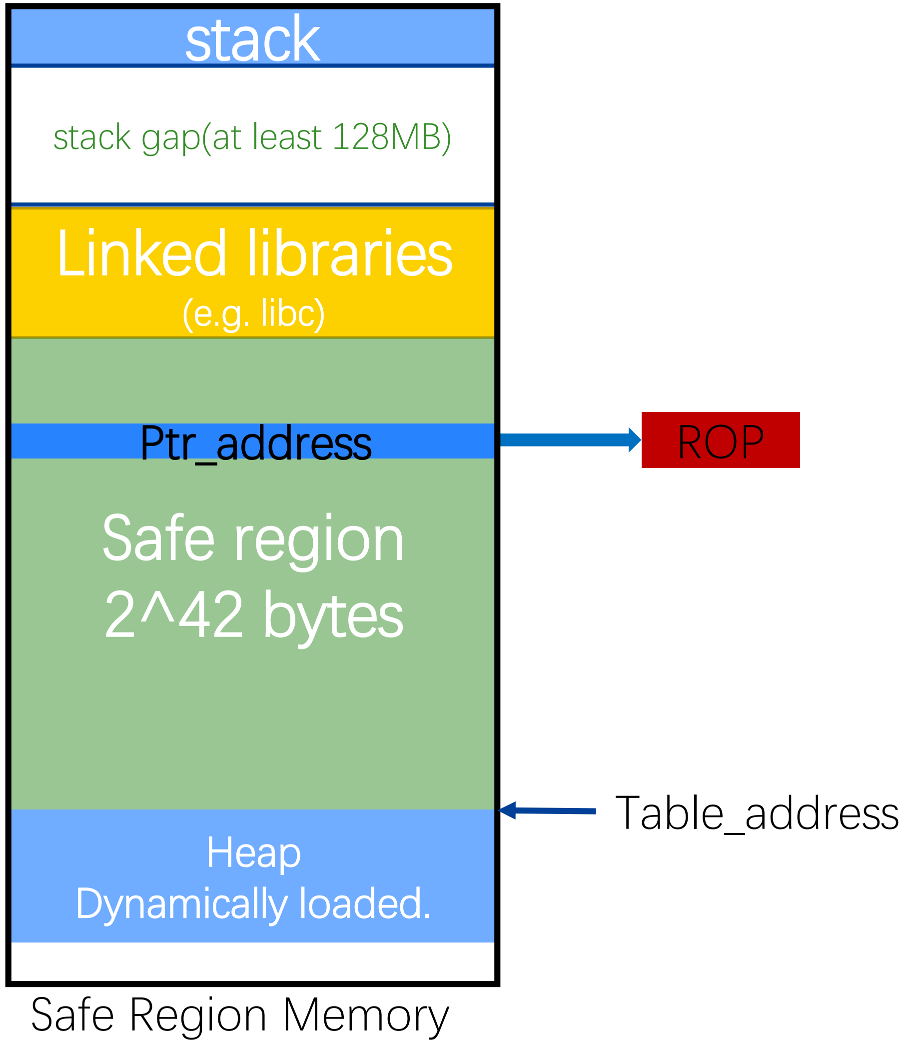
$$table\_address = libc\_base - 2^{42}$$



Safe Region Memory

3) Attack Safe Region

- Using the safe region table address, the address of a code pointer of interest in the CPI protected application, ptr_address, can be computed by masking with the cpi_addr_mask, which is 0x00fffffffff8, and then multiplying by the size of the table entry, which is 4.
- Armed with the exact address of a code pointer in the safe region, the value of that pointer can be hijacked to point to a library function or the start of a ROP chain to complete the attack.



4

PART FOUR

Measurements & Results

Measurements And Results

- Configuration
  - Nginx 1.6.2
  - compiled with clang/CPI 0.2 and the –flto –fcpi flags
  - 1 Gbit wired LAN connection
  - Intel i5 (39 bits physical address size, 48 bits virtual address size)
  - 4 GB RAM
- the attack measurements
  - Vulnerability
  - Timing Attack
  - Locate Safe Region
  - Fast Attack with Crashes
  - Attack Safe Region

Vulnerability

- Requirements
  - A parameter used as the upper loop bound
  - allowing user to gain control of the parameter by stack buffer overflow
  - launch the vulnerability over a wired LAN connection or wireless networks

Nginx logging module

nginx_http_parse.c

```
for (i = 0; i < headers->nelts; i++)
```

Timing Attack

- Measurements
  - Measuring the HTTP request round trip time(RTT) for a static web page(0.6KB) using Nginx
  - Collect 10000 samples to establish the average baseline delay

Timing Attack

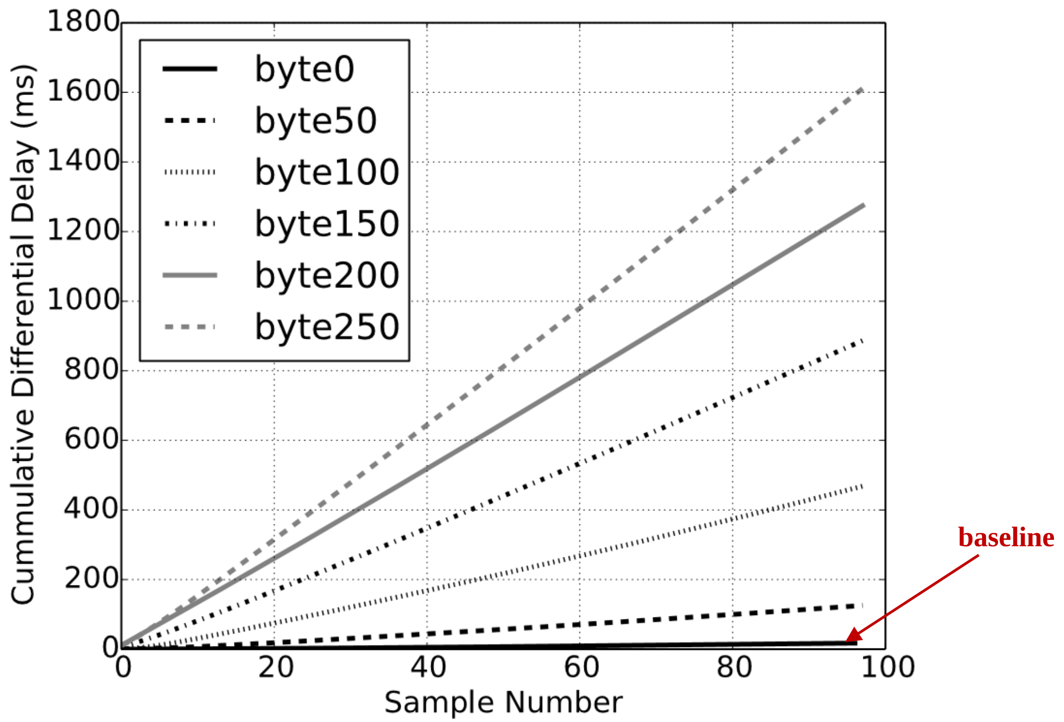


Fig. 4. Timing Measurement for Nginx 1.6.2 over Wired LAN

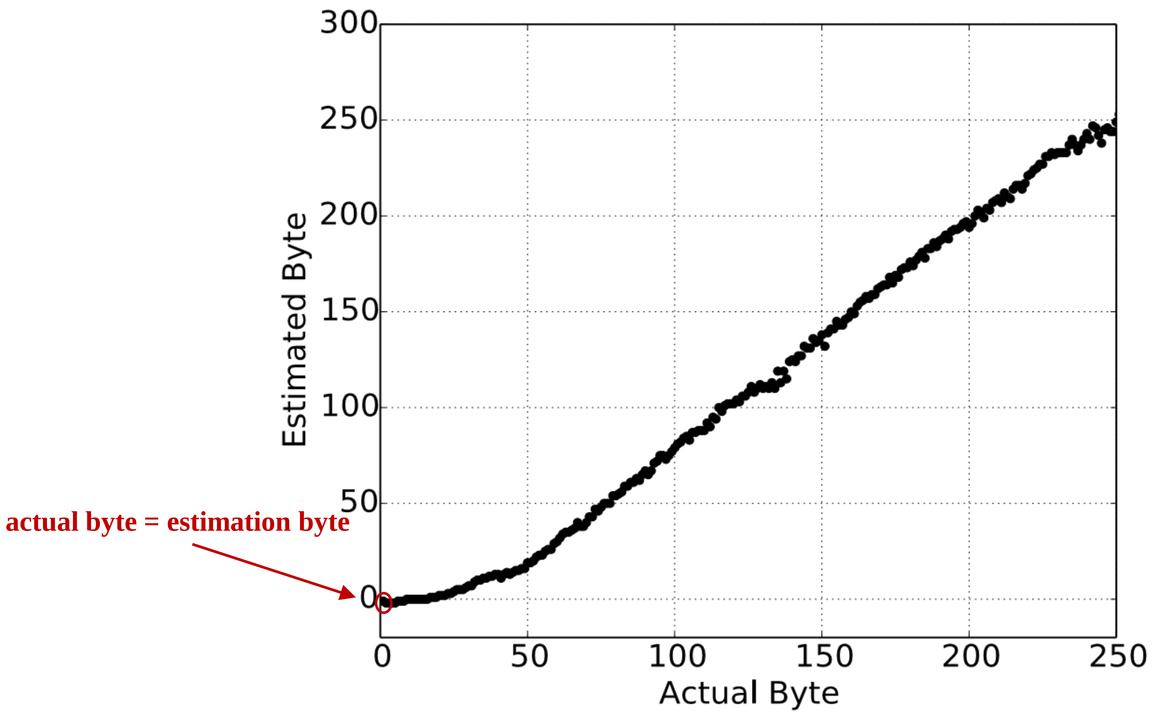


Fig. 5. Observed Byte Estimation

- **Results** (see Figure 4 and Figure 5)
  - The average RTT is 3.2 ms (**baseline**)
  - The byte estimation is accurate to within 2% (±20)

Locate Safe Region

- Problems
  - Problem 1 : How to determine whether a given page lies inside the safe region or inside the linked libraries ?
    - case 1 : hit a nonzero address inside the safe region, causing a false positive
    - case 2 : sample zero bytes values while inside libc
  - Problem 2 : How to identify our likely location in libc ?
    - low accuracy of identifying nonzero value's actual value
    - matching algorithm

Locate Safe Region

- Solution for Problem 1
  - Solution for case 1

    1. In tests, every byte read from the high a

  - Solution for case 2

    1. (libc_page_bytes[1272] == nonzero)
    2. using 30 sample per byte, scanning 5

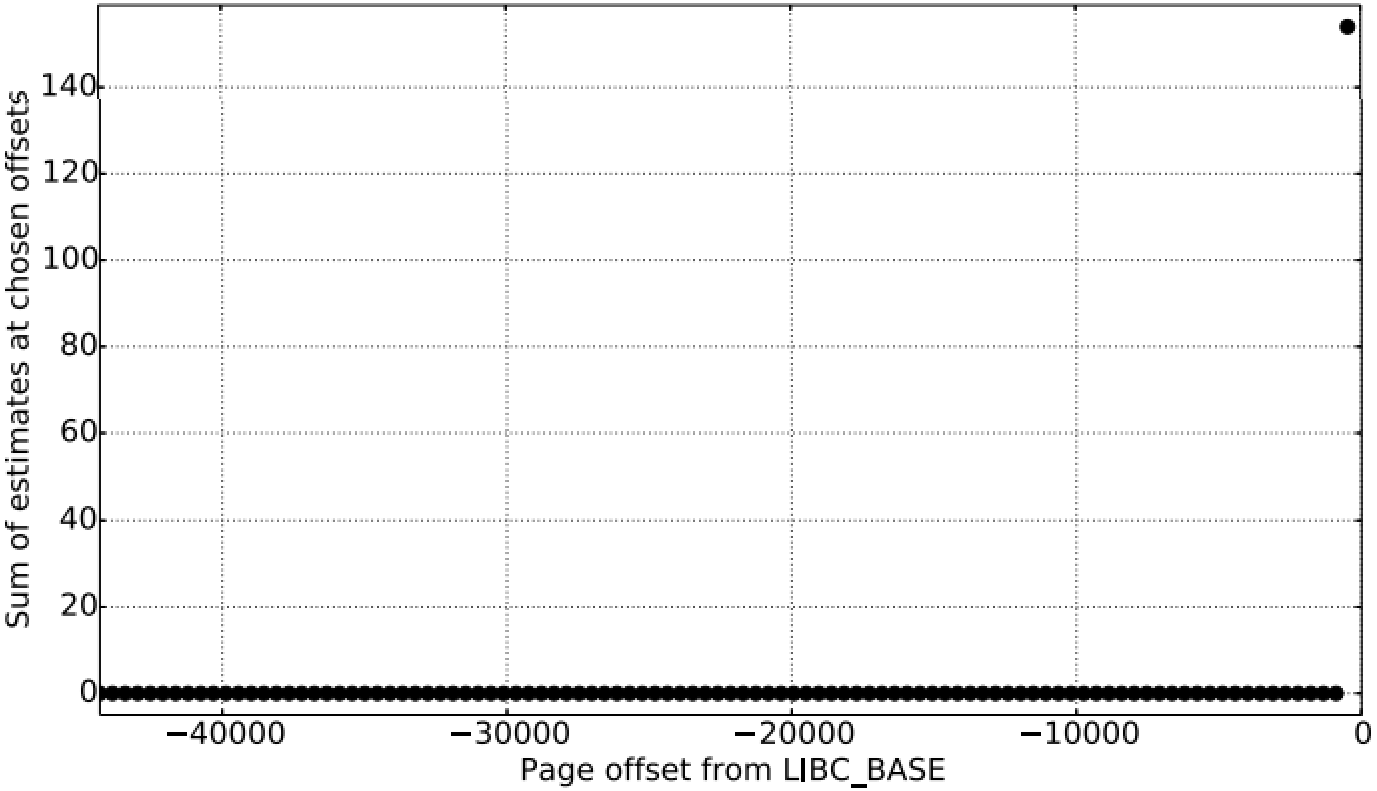


Fig. 6. Estimation of Zero Pages in Safe Region.

Figure 6 illustrates the sum of the chosen offsets for the scan of zero pages leading up libc

Locate Safe Region

- Solution for Problem 2

1. achieve high accuracy by sending 10000 samples per byte

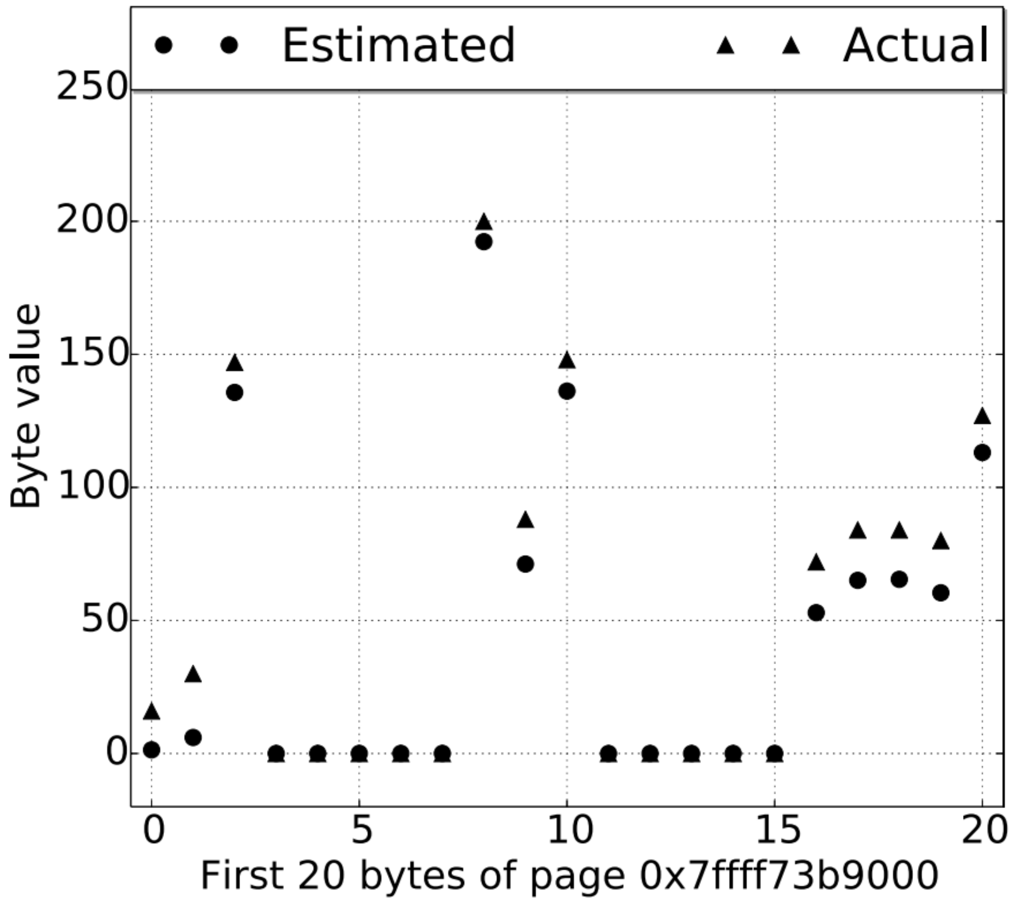
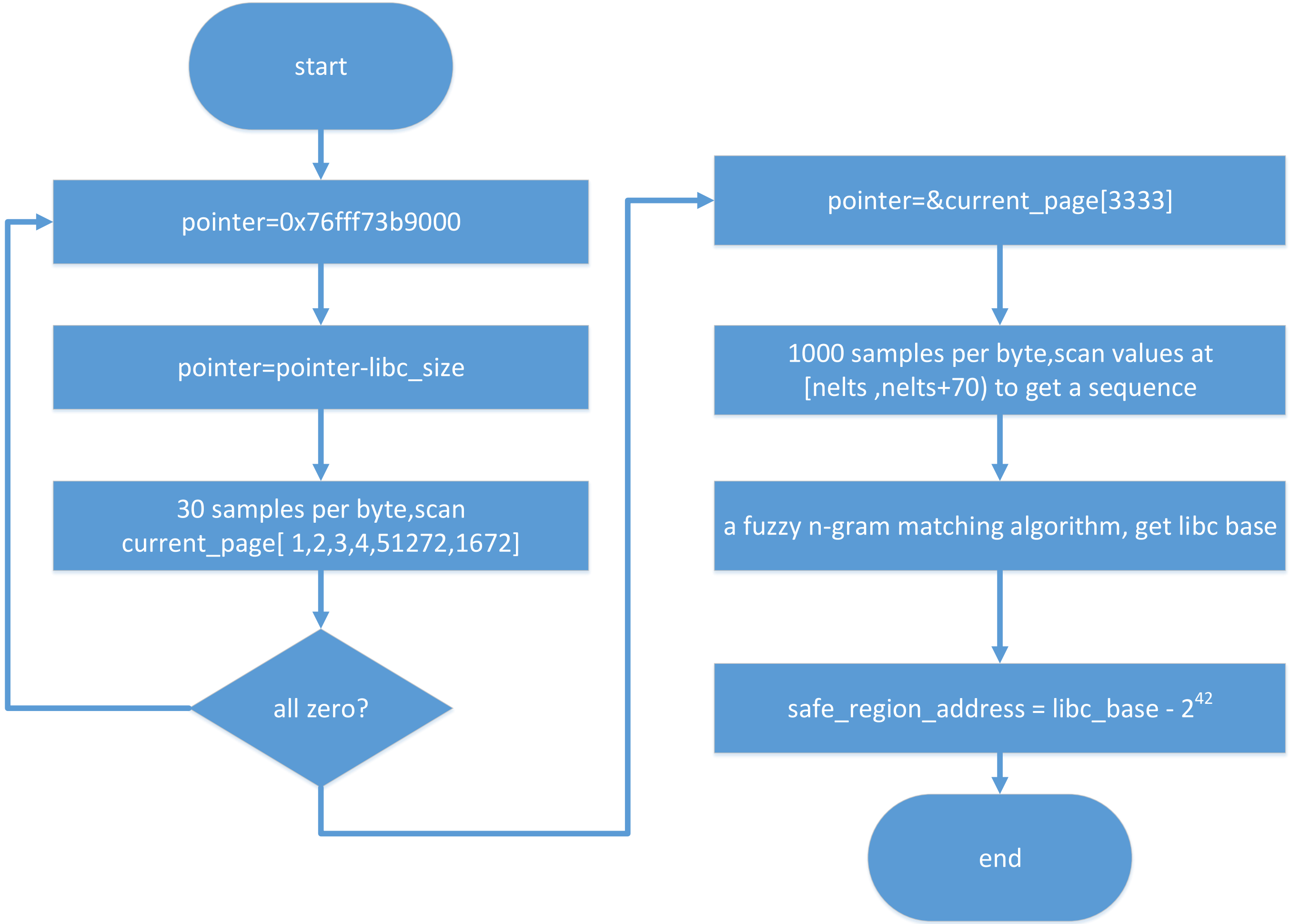


Fig. 7. Actual Bytes Estimation of a Nonzero Page in LIBC.

Locate Safe Region

locating process:



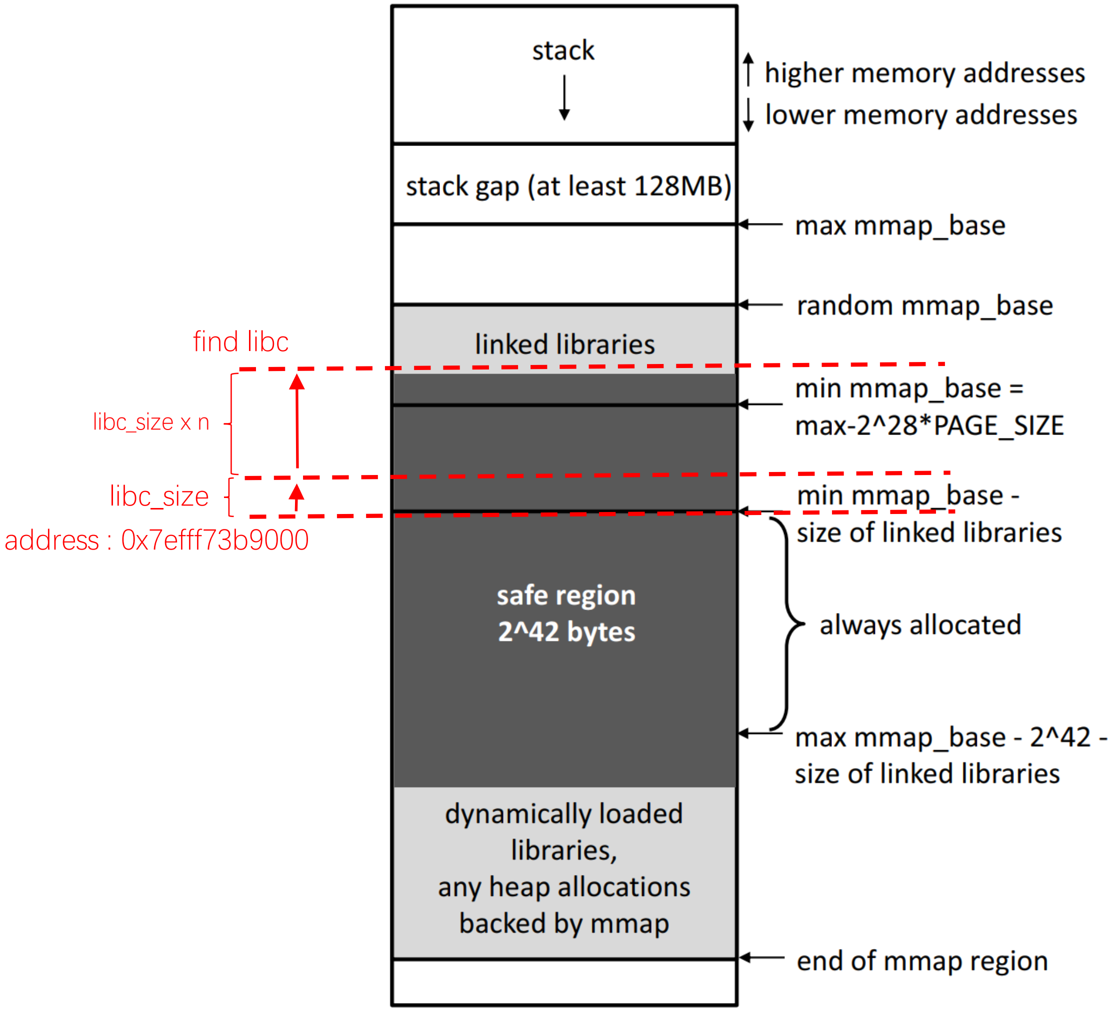


Fig. 1. Safe Region Memory Layout.

Fast Attack with Crashes

- The binary searchcaused 11 crashes
- Discovering the base of libc required an additional 2 crashes

Attack Safe Region

ngnix execution:

…

call read_handler_index

…

Address
Empty Slot
Lower Bound
Upper Bound

…

function read_handler…

…

function sysenter

…

ring 3 → ring 0

ROP Attack

→ do everything

5

PART FIVE

Countermeasures

Possible Countermeasures

- Memory Safe
  - low overhead mechanisms, such as approximate or partial memory safe
  - hardware support, Intel memory protection extensions(MPX)
- Randomization
  - continuously rerandomize the safe region

trade off security

- Timing Side Channel Defense
  - every execution takes the same amount of time

trade off performance

Discussion

- Design Assumptions
  - Enforcement Mechanisms
    - Not enough protection for the enforcement mechanisms
  - Detecting Crashes
    - A large number of pages are allocated
  - Memory Disclosure
    - Indirect leaks using dangling data pointers and timing or fault analysis attack
  - Memory Isolation
    - Random searching of the mmap region can used to leak the safe region

Discussion

- Patching CPI
  - Increase Safe Region Size
  - Randomize Safe Region Location
  - Use Hash Function For Safe Region
  - Reduce Safe Region Size
  - Use Non Contiguous Randomized mmap

THANK YOU！

Q & A