HeapTherapy+: Efficient Handling of (Almost) All Heap Vulnerabilities Using Targeted Calling-Context Encoding

Qiang Zeng,
Golam Kayas, Emil Mohammed, Lannan Luo,
Xiaojiang Du, and Junghwan Rhee

DSN 2019

UNIVERSITY OF SOUTH CAROLINA

NEC Laboratories America
Relentless passion for innovation

Trend of Memory Vulnerability Exploitation

- Memory vulnerability exploitation
  - Stack-based
  - Heap-based
- Many effective protection for call stacks
  - Stack canaries
  - Reordering local variables
  - Safe SEH (Structured Exception Handling)
- Heap vulnerability exploitation becomes the trend
  - **Heartbleed**: heap buffer overread
  - **WannaCry**: heap buffer overwrite
  - Popular **ROP (return oriented programming) attack** [1]:
    Heap overflow => overwrite a function pointer => **stack pivoting**

[1] McAfee, “Emerging ‘Stack Pivoting’ Exploits Bypass Common Security”, 2013

"Because the success of stack-based exploits has been reduced by the numerous security measures, heap-based attacks are now common" **[Ratanaworabhan 2009]**

[Ratanaworabhan 2009] Ratanaworabhan, et al.."NOZZLE: A Defense Against Heap-spraying Code Injection Attacks." *USENIX Security*. 2009.

Types of Heap Vulnerabilities

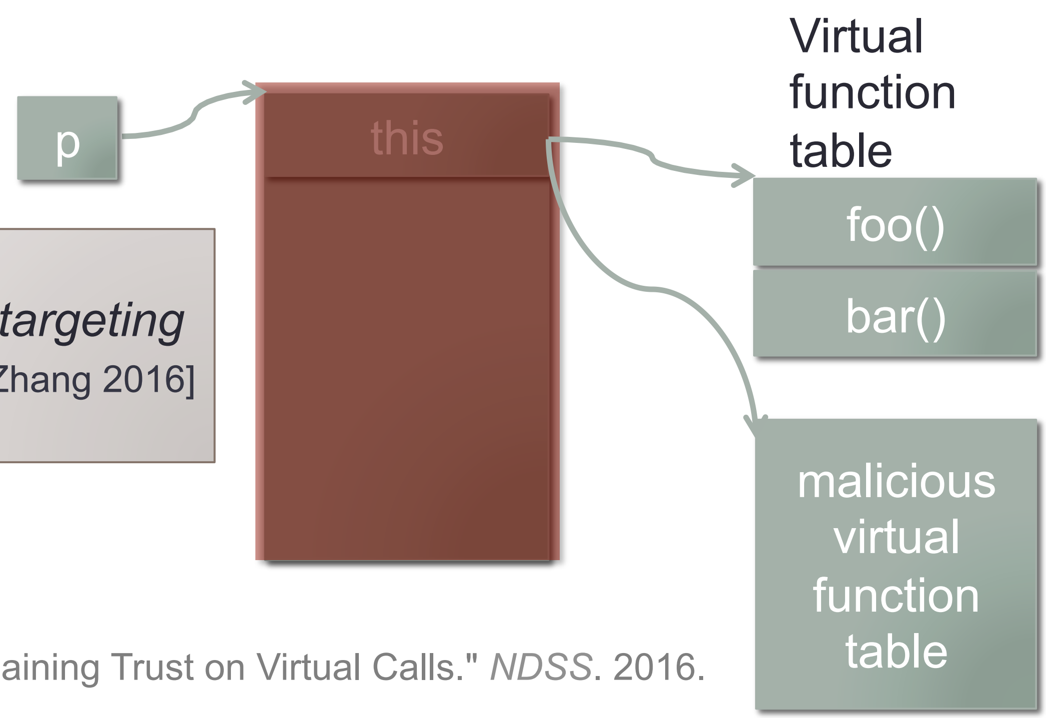
- Uninitialized read
  - Information leakage; …

```
str = (char*) malloc(128);
… // str is not initialized
cout << str;
```

Types of Heap Vulnerabilities

- Uninitialized read
  - Information leakage; …
- Use-after-free
  - Control-flow hijacking; …

```
(1) D *p = new D();
…
(2) delete p;
(3) …// buffer re-allocated and used
(4) p->foo(); // use-after-free
```



> *"More than 50% of known attacks targeting Windows 7 exploit use-after-free"* [Zhang 2016]

[Zhang 2016] Zhang, Chao, et al. "VTrust: Regaining Trust on Virtual Calls." *NDSS*. 2016.

Types of Heap Vulnerabilities

- Uninitialized read
  - Information leakage; …
- Use-after-free
  - Control-flow hijacking; …
- Buffer overflow
  - Over-write
    - Manipulating data; control-flow hijacking; …
  - Over-read
    - Information leakage; …

Existing Measures

- Checking every buffer access is great…but expensive
  - SoftBound (handle overflow and use-after-free): 67%
  - AddressSanitizer (handle overflow and use-after-free): 73%
  - MemorySanitizer (handle uninitialized read): 2.5x
- SFI (software fault isolation), CFI (control-flow integrity), XFI, CPI (code pointer integrity), …
  - Challenges when working with existing shared libs (legacy code)
  - Some (like XFI) are still very expensive
- Our previous work
  - Cruiser [PLDI'11], Kruiser [NDSS'12]: only handle overwrite
  - HeapTherapy [DSN'15]: only handle overwrite and overread

A Patching Perspective

- Patching is an indispensable step throughout the life of a software system; however,
  - 153 days on average for delivering a patch [1]
  - Only 65% of vulnerabilities have patches available [2]
  - Fresh patches break systems frequently

- ***Our goals***
  - Handle heap **overflow, uninitialized read, and use-after-free**
  - Generate patches instantly with **zero** manual diagnosis efforts
  - Install patches without altering code, i.e., **code-less patching**
  - A **very small** overhead

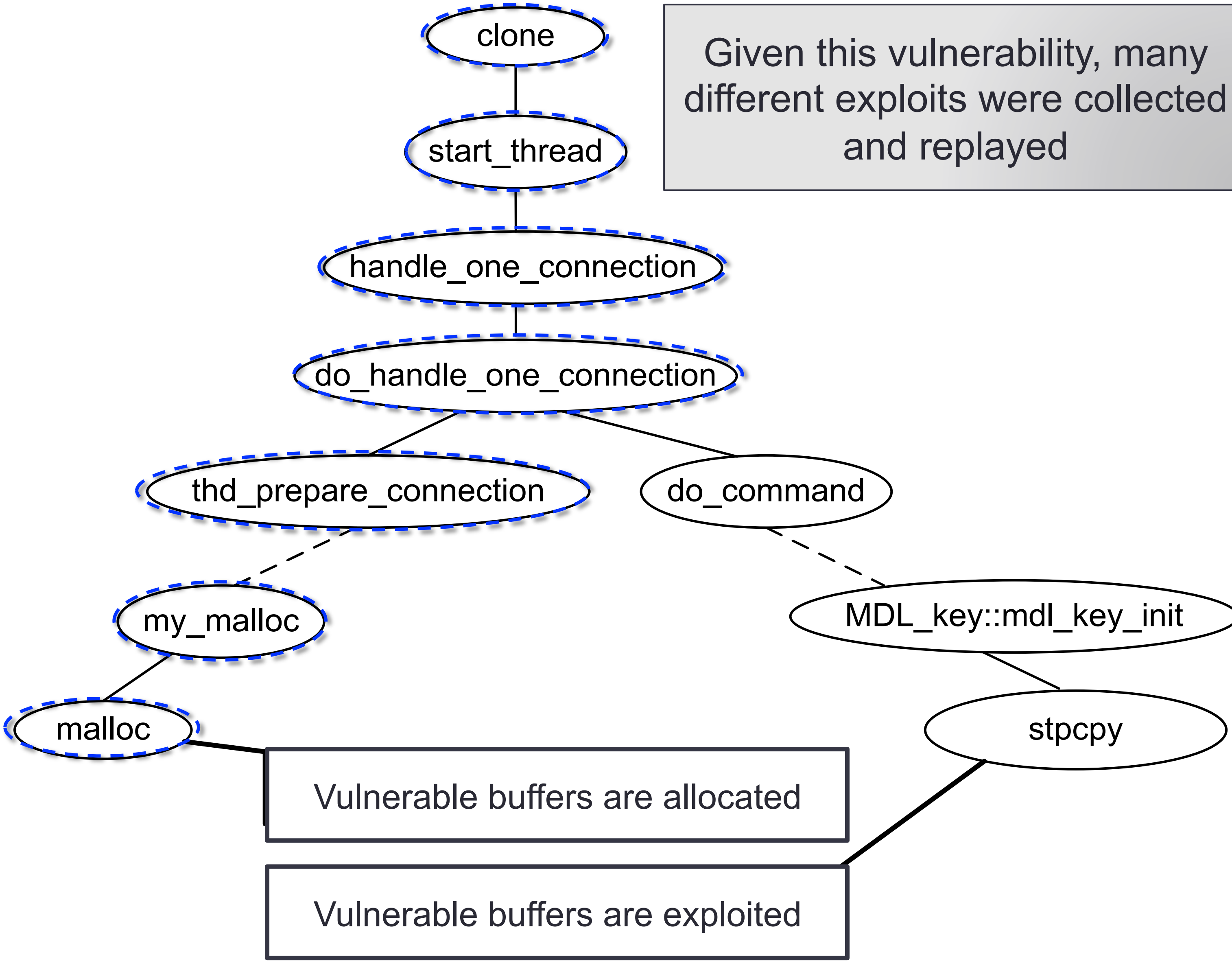
[1] S. Frei, "The Known Unknowns," 2013.
[2] S. frei, "" "End-point security failures, insight gained from secunia psi scans," 2011.

Hypotheses

Given a heap buffer overflow bug, the vulnerable buffers share the same calling context when they are *allocated*

More generally, for a use-after-free or uninitialized-read vulnerability, the vulnerable buffers share the same calling context when they are allocated

Verifying Hypotheses



Given this vulnerability, many different exploits were collected and replayed

Main Approach

Using **allocation-time calling context** to characterize vulnerable buffers

1. When replaying the attack, record the allocation-time calling context of each buffer. When the offending operation (e.g., overflow) is detected, output the allocation-time calling context of the vulnerable buffer

2. During runtime, if a buffer being allocated has that allocation-time calling context, enhance it

Challenges

- How to retrieve and compare calling contexts efficiently?
  - Retrieving calling context via stack walking is too **expensive**
  - ASLR makes the collected RAs **useless**
- How to bridge offline attack analysis and online defense generation?
- How to achieve code-less patching?
- How to handle the diverse vulnerabilities in a uniform way?

- **Targeted Calling Context Encoding**
- **Offline Attack Analysis and Patch Generation**
- **Online Defense Generation**

Calling Context Encoding

- Using an integer (or very few integers) to encode the calling context
  - The integer is updated at each function *call* and *return*; using simple arithmetic operations
- <3% slowdown; concise representation
- Wide applications: testing coverage, anomaly detection, compilation optimization, logging, …

	PCC [Bond 2007]	PCCE [Sumner 2010]	DeltaPath [Zeng 2014]
Support Object-Oriented	✔	✗	✔
Decoding	✗	✔	✔
Scalability	✗	✗	✔

Example: PCC

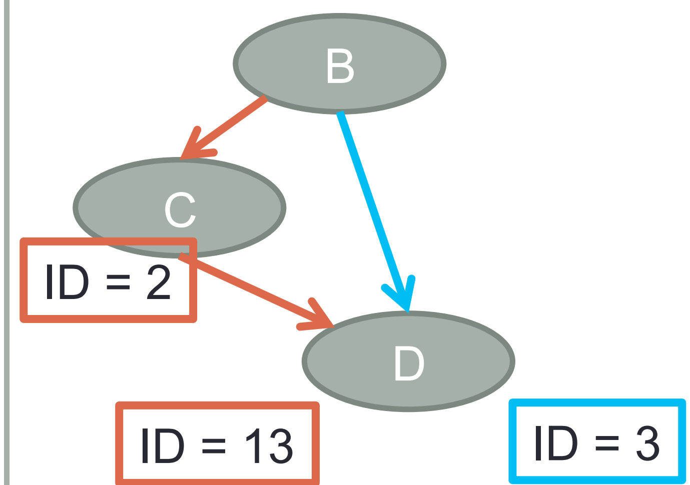
- Goal: each unique ID represents a unique calling context

```
ID = 0

B() {          // t = ID
 C();          // ID = t * 3 + 2
 D();          // ID = t * 3 + 3
}

C() {          // t = ID
D();           // ID = t * 3 + 7
}

D() {
 Sensitive API! // calling context?
}
```



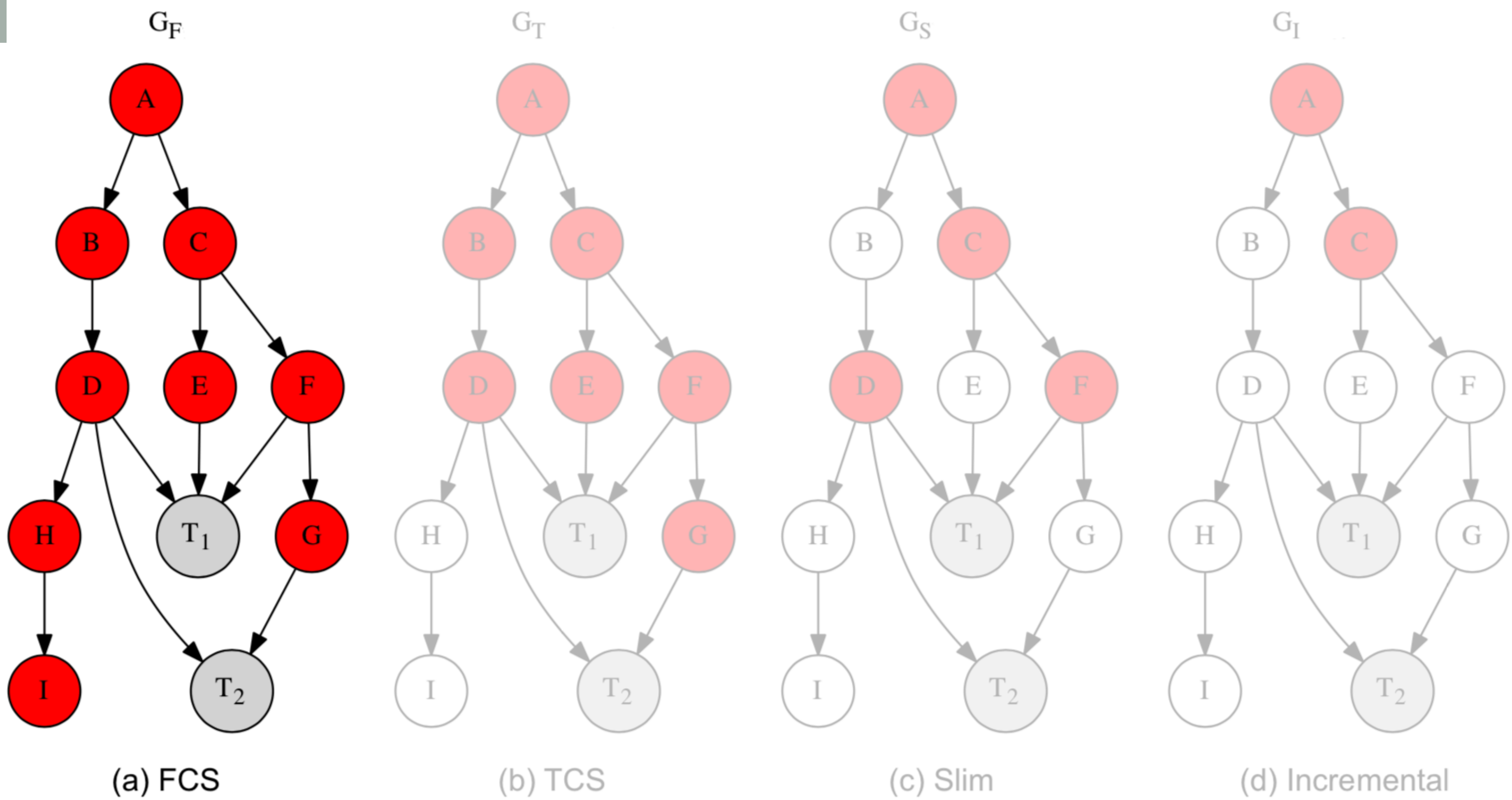
Answer: Read the variable “ID” to get the calling context ID

Targeted Calling Context Encoding

- A set of ideas that can minimize the encoding overhead

- **Key insight**: When the *target functions*, whose calling contexts are of interest, are known, many call sites do no need to be instrumented
  - E.g., some functions never appear in the calling contexts that lead to the target functions

- Target functions in our work:
  - malloc, calloc, realloc, memalign, aligned_alloc



(a) FCS (full-call-site instrumentation): original PCC encoding

(b) TCS (targeted-call-site): **H** and **I** cannot reach the targets **T1** and **T2**

(c) Slim: **B**, **E** and **G** each has only one out-going edge that reaches the targets

(d) Incremental: **F-T_1** and **F-G-T_2** can be distinguished through the target

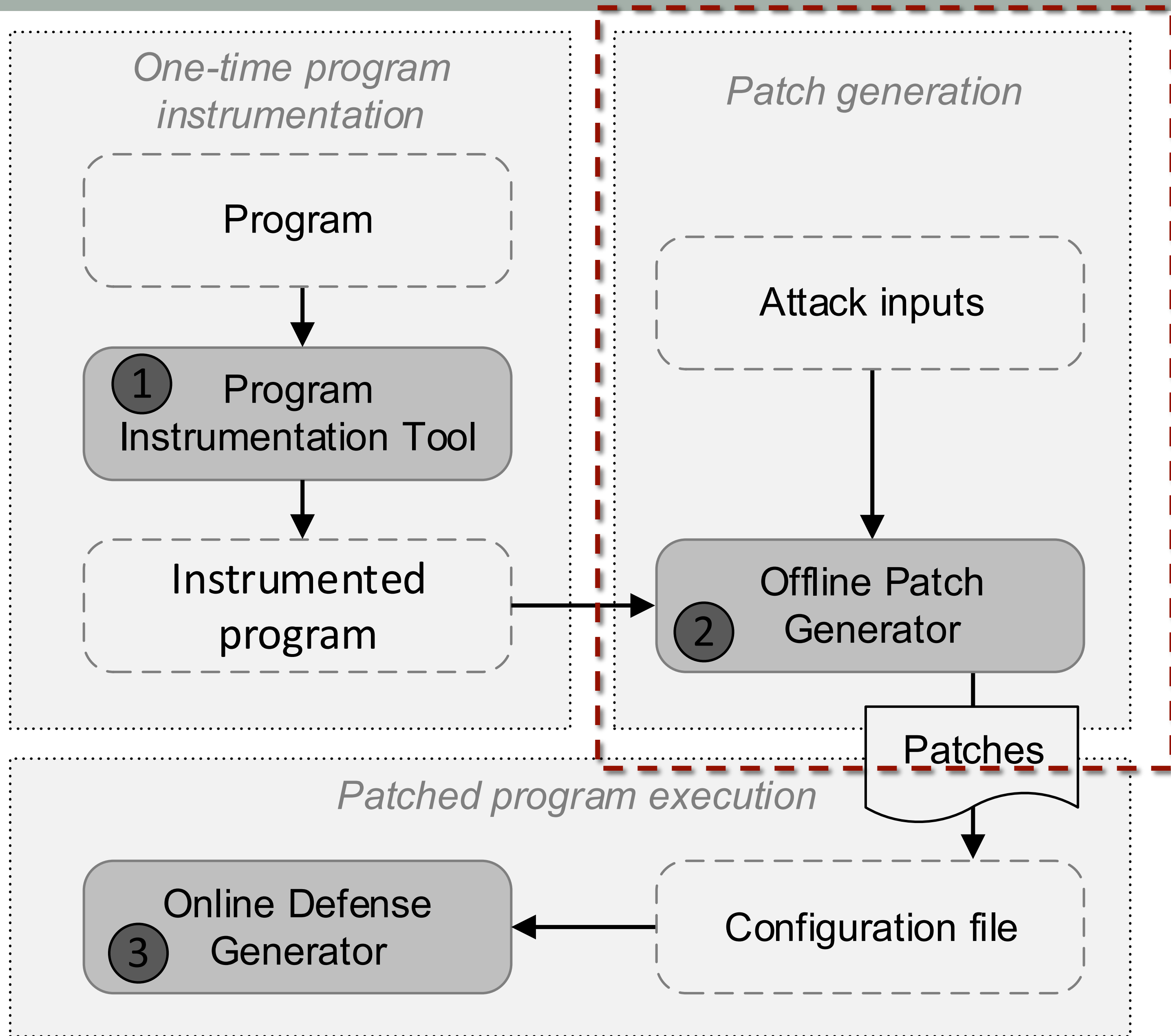
Encoding overhead

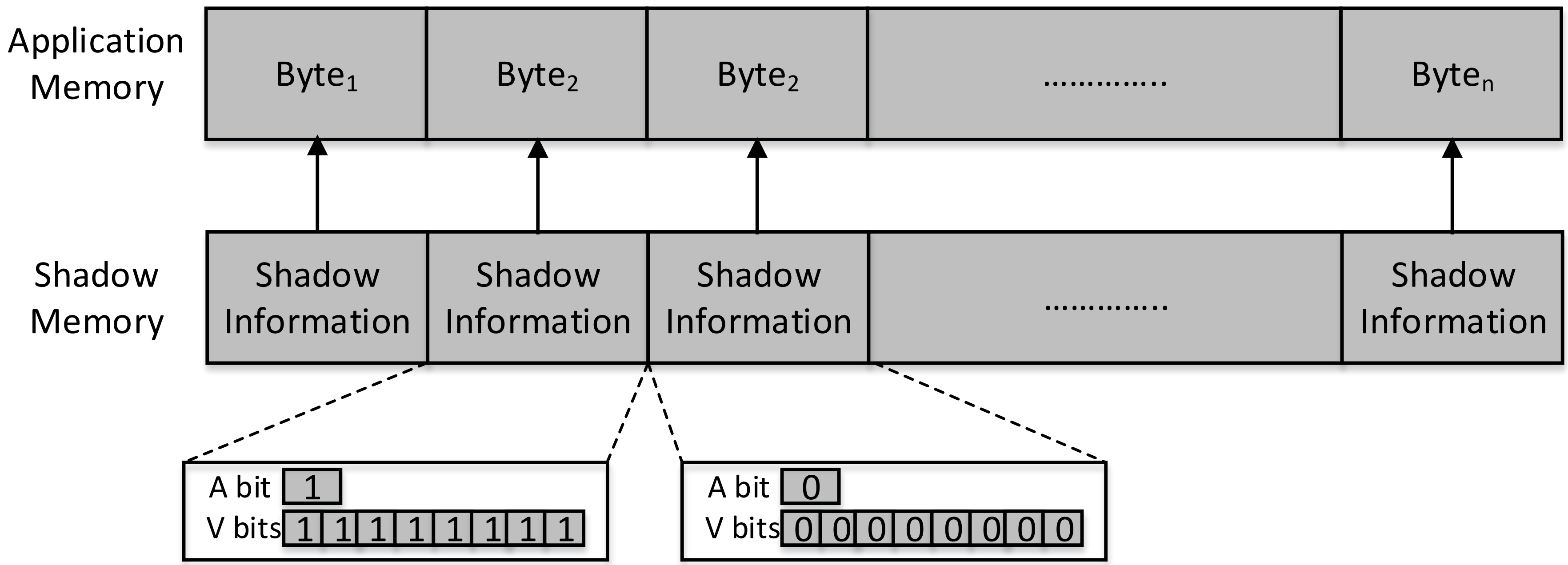
- Implementation: added an LLVM pass for instrumentation
- Evaluation: SPEC CPU2006 Integer

- Size overhead
  - PCC: 12%
  - Targeted Calling context Encoding: 4.4%
  - 2.7x of improvement

- Speed overhead
  - PCC: 2.4%
  - Targeted Calling Context Encoding: 0.4%
  - **6x of speed up**

- **Targeted Calling Context Encoding**
- **Offline Attack Analysis and Patch Generation**
- **Online Defense Generation**


One-time program instrumentation
Program
1 Program Instrumentation Tool
Instrumented program
Patch generation
Attack inputs
Offline Patch Generator 2
Patches
Patched program execution
Online Defense Generator 3
Configuration file



- Accessibility-bit (A-bit): whether the byte can be accessed
  - If a buffer has been free-ed, all its A-bits are 0
  - Each buffer is surrounded by two red zones (16B each), whose A-bits are 0
- Validity-bit (V-bit): whether the bit is initialized
  - When a fresh buffer is malloc-ed, all it V-bits are 0
- Each buffer's alloc-API and CCID are recorded

(1) Detect overflow: an overflow will touch the inaccessible red zone
(2) Detect use-after-free: a free-ed buffer is set as inaccessible and then added to a queue to delay the space reuse
(3) Detect uninitialized read: more complex, but mainly relies on V-bits

Patches as a configuration file

- Each patch is simply a tuple

  <alloc-API, CCID, vul-type>

- Code-less patching: to “install” a patch, just add one line in the config file

Configuration file

```
<API,        CCID,         Vulnerability>
-------------------------------------------
<memalign, 1854955292,  OVERFLOW>
<calloc,     8643565443,  USE-AFTER-FREE>
<malloc,     2598251483,  UNINITIALIZED-READ>
… ...
```

- **Targeted Calling Context Encoding**
- **Offline Attack Analysis and Patch Generation**
- **Online Defense Generation**

Patches read into a hash table

Configuration file

<API,	*CCID,*	*Vulnerability>*
<memalign,	1854955292,	OVERFLOW>
<calloc,	8643565443,	USE-AFTER-FREE>
<malloc,	2598251483,	UNINITIALIZED-READ>
… ...		

Read by *Online Defense Generator*

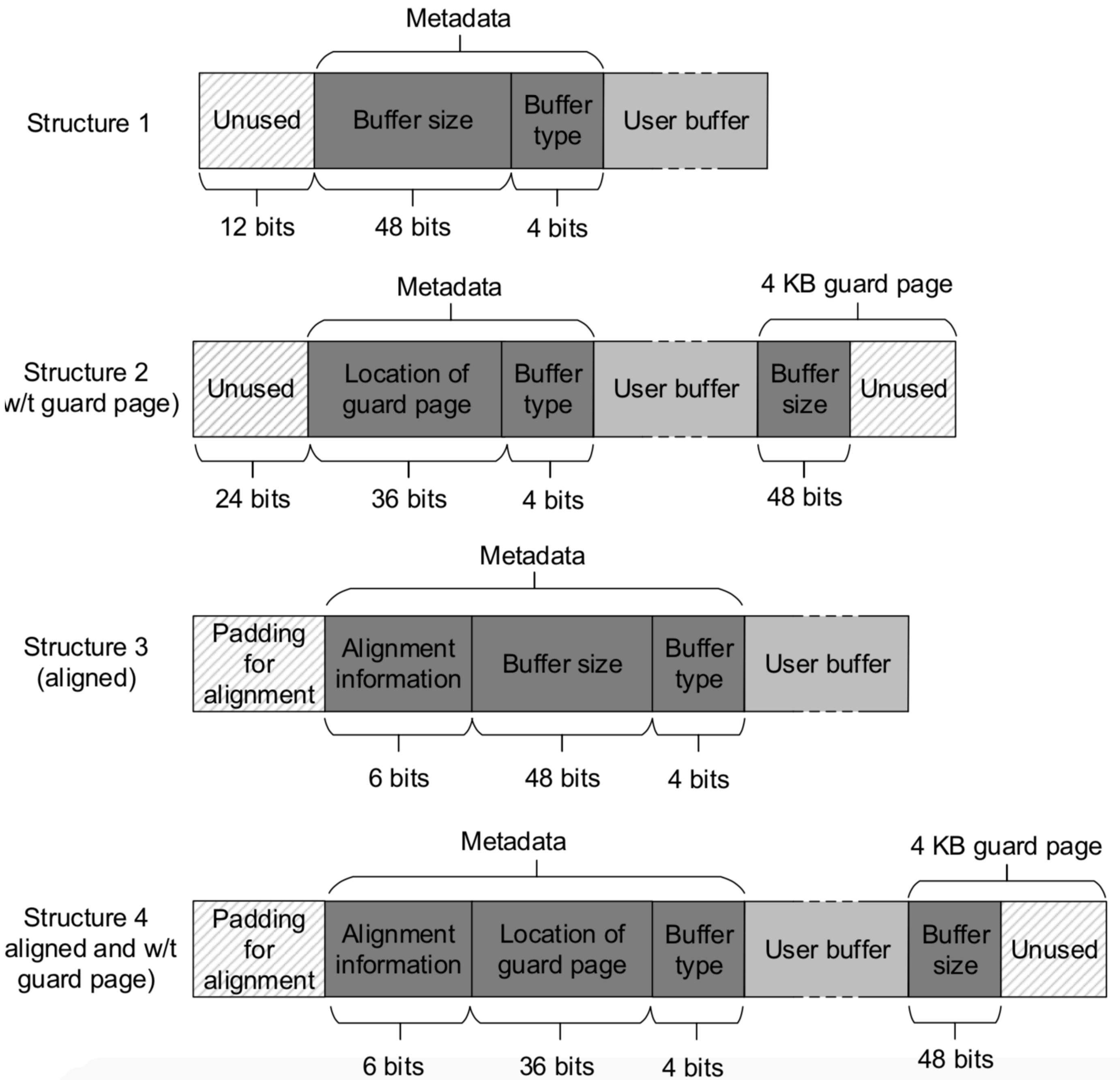
A shared lib

Hash table

Key	*Value*
<MEMALIGN, 1854955292>	$(001)_2$
<CALLOC, 8643565443>	$(010)_2$
<MALLOC, 2598251483>	$(100)_2$
…..	… …

Vulnerability Handling

- Handle overflow
  - Append a guard page to each vulnerable buffer
- Handle use-after-free
  - Delay the deallocation of the free-ed vulnerable buffers
- Handle uninitialized read
  - Initialize the newly allocated vulnerable buffer with zeros


Metadata
Structure 1
Unused
Buffer size
Buffer type
User buffer
12 bits
48 bits
4 bits
Metadata
4 KB guard page
Structure 2 (w/t guard page)
Unused
Location of guard page
Buffer type
User buffer
Buffer size
Unused
24 bits
36 bits
4 bits
48 bits
Metadata
Structure 3 (aligned)
Padding for alignment
Alignment information
Buffer size
Buffer type
User buffer
6 bits
48 bits
4 bits
Metadata
4 KB guard page
Structure 4 aligned and w/t guard page)
Padding for alignment
Alignment information
Location of guard page
Buffer type
User buffer
Buffer size
Unused
6 bits
36 bits
4 bits
48 bits

Evaluation

- Effectiveness

Program	Vulnerability	Reference
Heartbleed	UR & Overflow	CVE-2014-0160
bc-1.06	Overflow	Bugbench [57]
GhostXPS 9.21	UR	CVE-2017-9740
optipng-0.6.4	UaF	CVE-2015-7801
tiff-4.0.8	Overflow	CVE-2017-9935
wavpack-5.1.0	UaF	CVE-2018-7253
libming-0.4.8	Overflow	CVE-2018-7877
SAMATE Dataset	Variety	23 heap bugs [58]

- Efficiency
  - SPEC CPU2006: 4.3% (zero patch), 4.7% (one patch), 5.2% (five)
    - 1.9% due to malloc/free hooking, 2% due to buffer metadata maintaining
    - The 3.9% can be eliminated if our system is integrated into the allocator
  - MySQL (w/t Heartbleed): mysql-stress-test.pl; no observable overhead
  - Nginx (w/t Heartbleed): AB; throughput overhead 4.2%

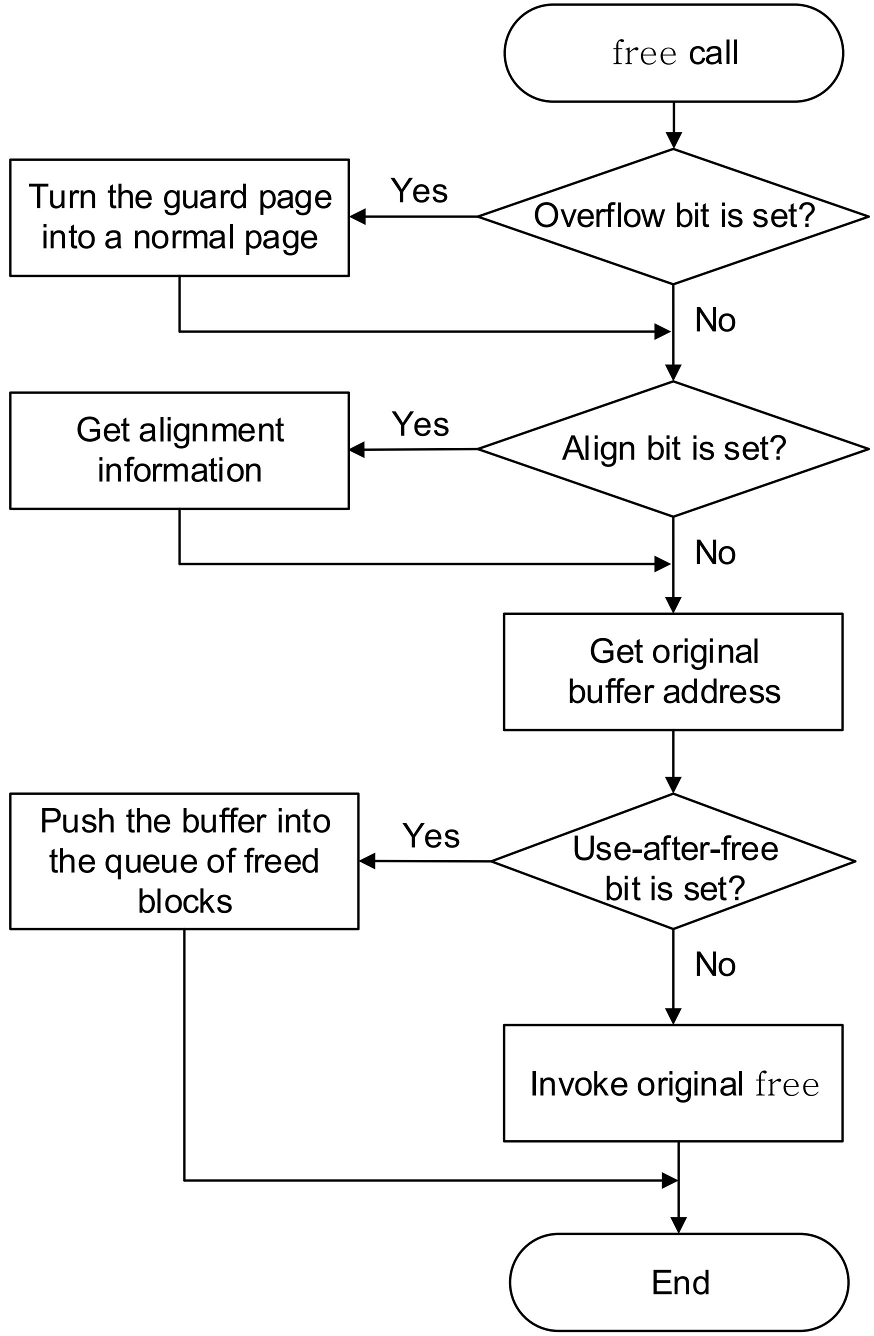
Contribution and Limitations

- The first work that can patch all the following heap vulnerabilities without manual analysis effort
  - **Overflow, use after free, uninitialized read**
- Prominent features:
  - Code-less patching
  - Very small overhead (several percentages)
  - You can still use your favorite heap allocator
- A showcase how heavyweight offline analysis can be seamlessly combined with lightweight online defenses
- **Targeted calling context encoding: 6x speed up**
- Limitations
  - Cannot handle some vulnerabilities: e.g., an overflow within a struct
  - Overflow leads to DoS: padding may be considered, as used in HeapTherapy
  - Re-compilation needed: binary instrumentation is possible

THANKS!

Q&A

Qiang Zeng (zeng1@cse.sc.edu)


free call
Overflow bit is set?
Yes
Turn the guard page into a normal page
No
Align bit is set?
Yes
Get alignment information
No
Get original buffer address
Use-after-free bit is set?
Yes
Push the buffer into the queue of freed blocks
No
Invoke original free
End