2015 Network and Distributed System Security (NDSS)

Preventing Use-after-free with Dangling Pointers Nullification

Byoungyoung Lee et al.

Jinheon Lee, Jin Heo

DANGNULL

- C/C++과 같은 저수준 언어에서 Use-After-Free를 막기 위한 방법 제시
- 3만개의 Chrome browser unit-test에서 False-Positive 발견 되지 않음
- Static analysis + Dynmic(Runtime) analysis 융합
- 평균 오버헤드
  - 실행시간 : 80% 증가
  - 메모리 오버헤드: 200 ~ 472% 증가

Use-After-Free

- 메모리 버그 중 하나인 UAF(Use After Free)는 2024년도 CWE Top 25 중 8위를 기록할 정도로 꽤나 영향력 있는 취약점에 해당함

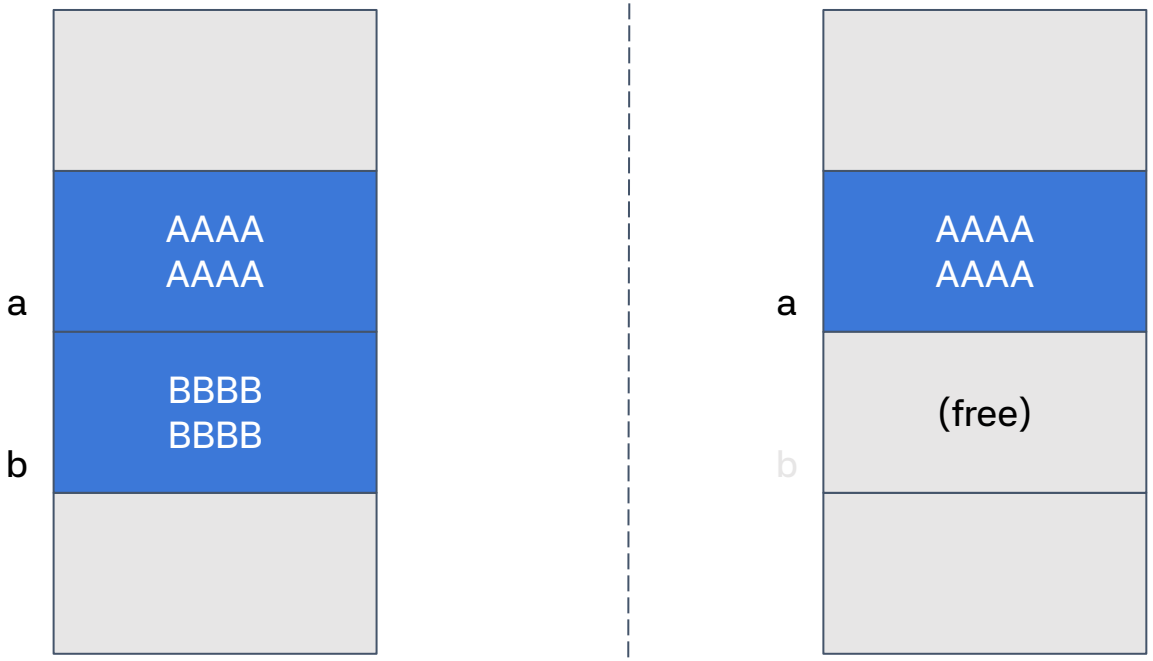
2024 CWE Top 25

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0

출처 : https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html#top25list

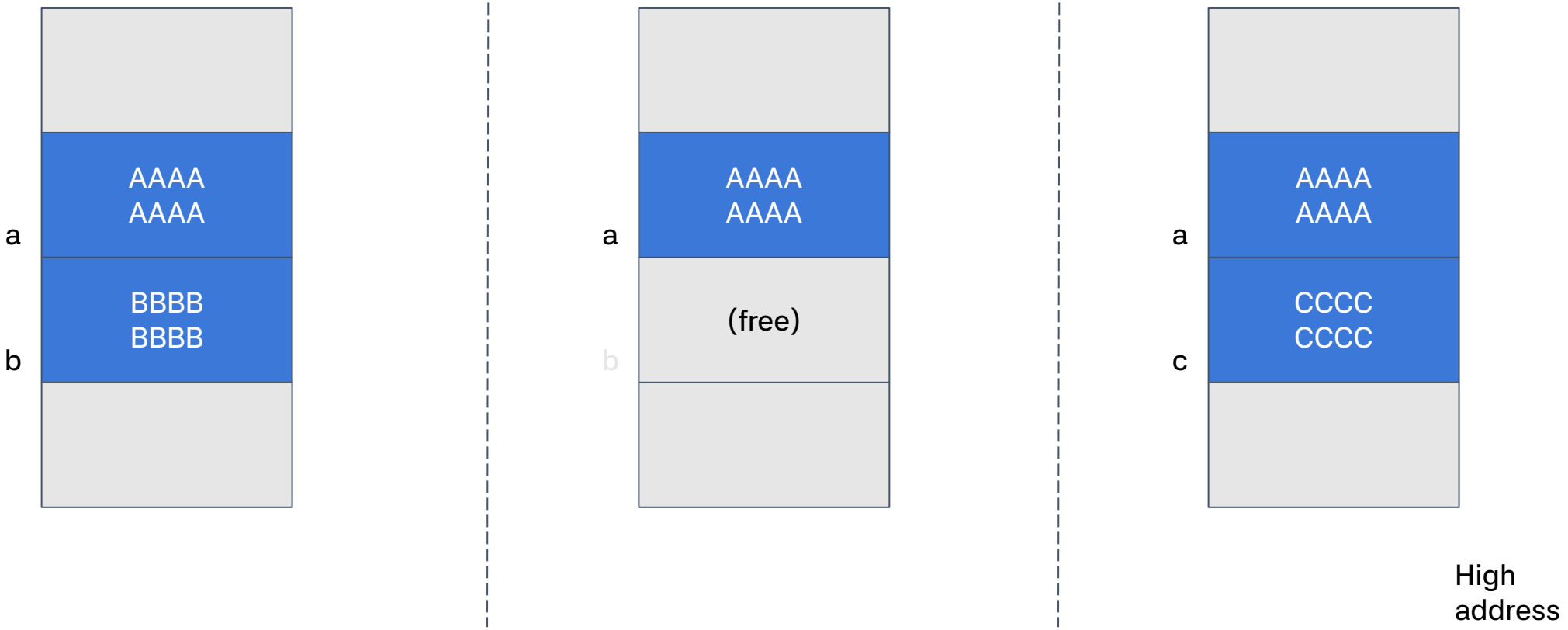
UAF (Use-After-Free)

UAF (Use-After-Free)

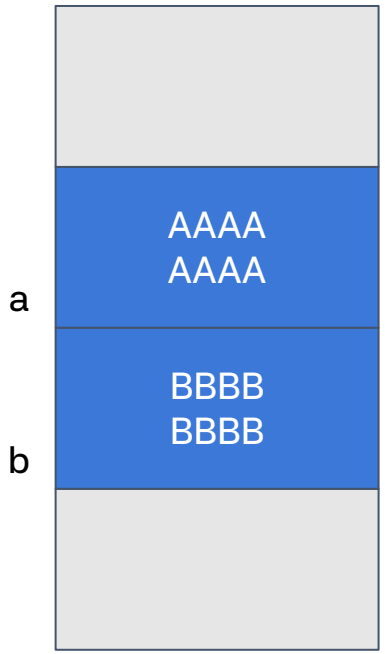


free(b);

UAF (Use-After-Free)

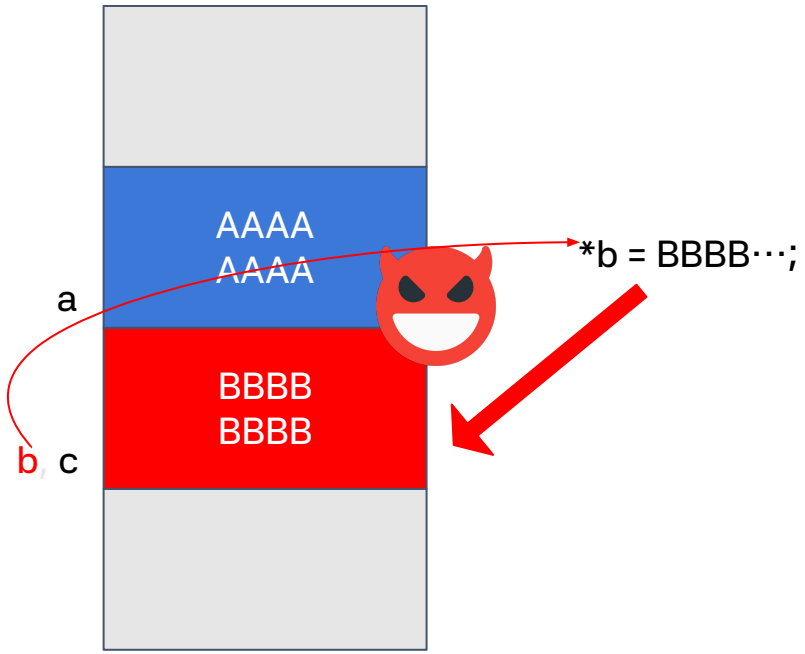


UAF (Use-After-Free)



free(b);

malloc(c);



From dangling pointer to UAF

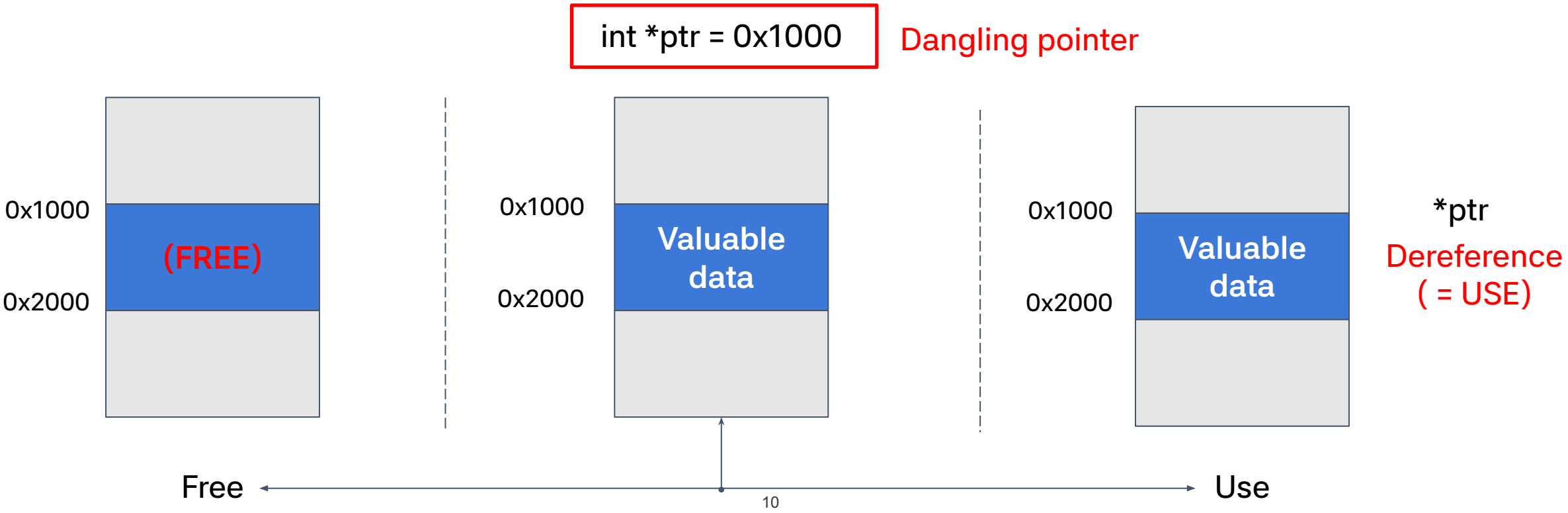
- Dangling pointer 자체로는 취약점이 될 수 없음.
- 해당 pointer에 접근해야 함.

Challenges in identifying dangling pointers

- 메모리의 할당, 해제, 확장, 사용은 다른 함수, 모듈에 분산되어있음.
- 이러한 operation이 여러 스레드에 걸쳐서 일어날 수 있음.

Exploiting dangling pointers

- Free된 영역에 가치 있는 데이터를 배치시키고, 이를 활용하는 것이 Exploit 방법.
- 즉, Free 와 Use 사이에 데이터 배치가 이루어져야한다.
- 데이터의 가치성에 따라 exploitability가 결정 됨.



Goal

- False-Positive, False-Negative 없이 Detect
- ⇒ Dangling pointer가 생기는 root cause 에 집중

Root cause of unsafe dangling pointer

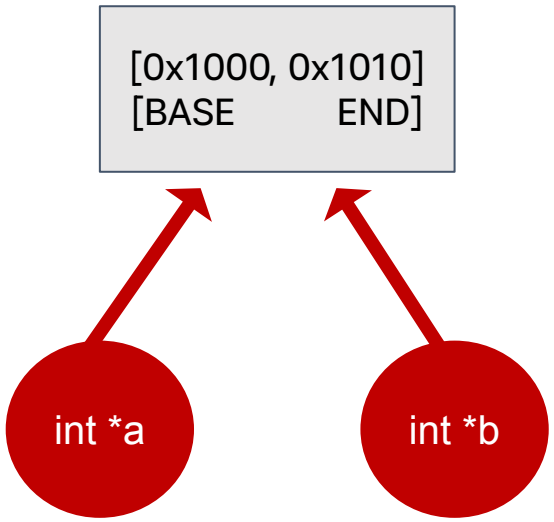
- 할당된 주소를 담은 initial 포인터가 다른 포인터 변수에 (연산을 하여) 대입 됨.
- 해당 할당된 주소가 free됐음에도, 전에 해당 주소가 대입된 포인터 변수들의 값은 초기화(NULL) 안됨.

```
int *a = malloc(0x30);
int *b = a+1;
free(a); a = NULL;
// However, `b` is referencing the free area (Non-Nullified), `b` is dangling pointer!
```

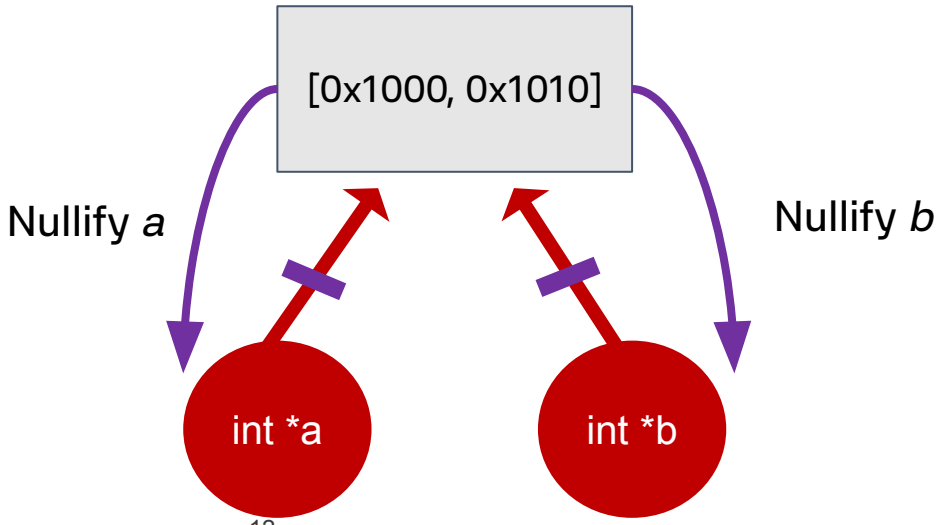
Overview

- Pointer와 할당된 memory object의 relationship을 추적
- object가 free될 경우, 해당 object를 가르키는 pointer들을 nullify

```
int *a = malloc(0x10); // 0x1000
int *b = a;
```




```
free(0x1000);
```



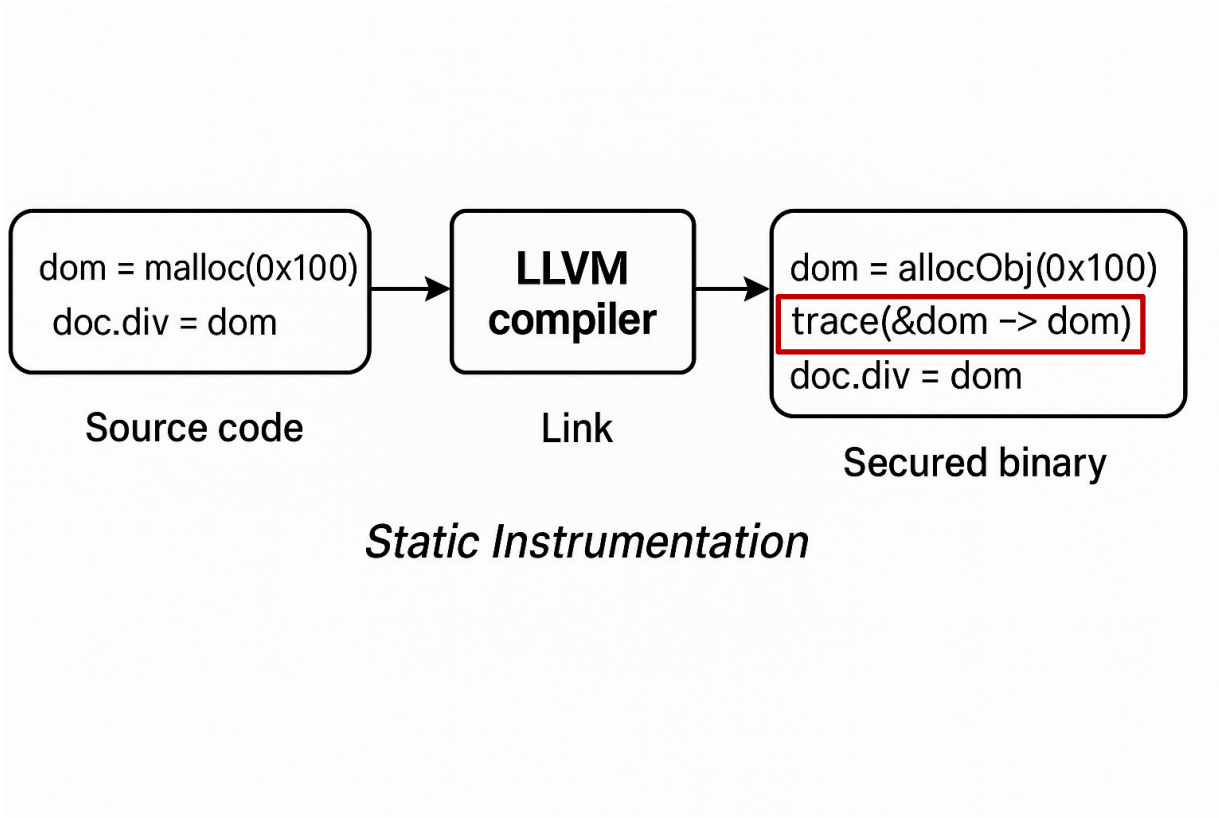
Result

a == b == NULL;

*a ⇒ ***SEGFAULT***

*b ⇒ ***SEGFAULT***

Overview



- DANGNULL은 Static Analysis + Runtime 을 모두 사용함.
- Compile단에서 Pointer assignment operation에 대해 *trace* 함수 삽입.
- Runtime단에서 trace 함수를 실행하여 pointer 추적하고 relationship-tree를 생성하게 됨

Static Instrumentation

```
int *a = malloc(0x10);
int *b = a;
int *c = b + 3;
```

Compiler

```
int *a = malloc(0x10);
int *b = a;
+ trace(a, b);
int *c = b + 3;
+ trace(a, b);
```

- Pointer assignments를 monitoring 하기 위해, LLVM을 이용하여 trace 함수 호출 삽입
  - 단, 모든 pointer assignment operations에 대해 삽입하지 않음. (다음 페이지 참고)

Static Instrumentation

- trace() 함수 대입 조건

```
for function in Program:
   # All store instructions are
   # in the LLVM IR form of 'lhs := rhs'.
   for storeInstr in function.allStoreInstructions:
      lhs = storeInstr.lhs
      rhs = storeInstr.rhs

      # Only insterested in a pointer on the heap.
      if mustStackVar(lhs):
         continue
      if not isPointerType(rhs):
         continue

      new = createCallInstr(trace, lhs, rhs)
      storeInstr.appendInstr(new)
```

int *a = b; // lhs: a , rhs = b

1. lhs 의 위치가 heap (≠ stack) 이어야 함.
   - Performance Overhead 를 줄이기 위함.
   - Stack var는 short lifetime을 갖기 때문에, 배제. (⇒ Limitation)

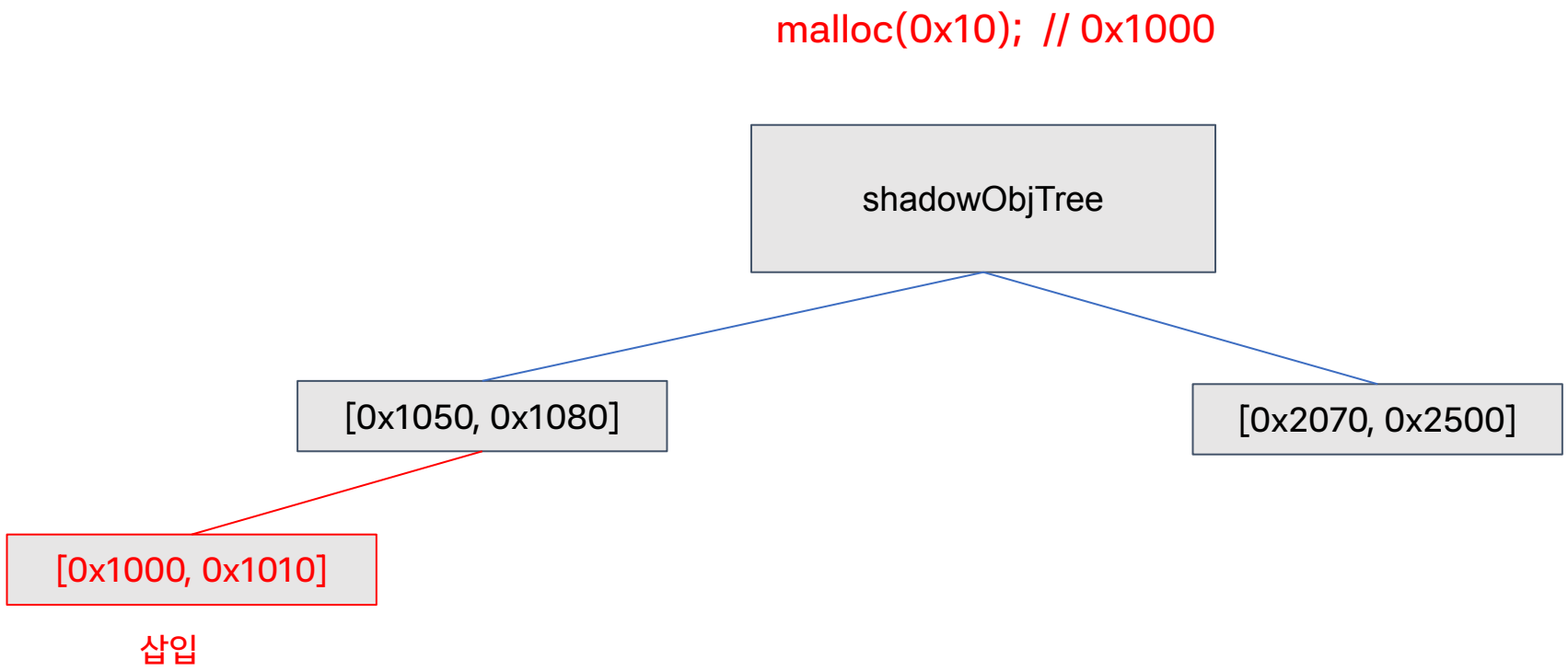
2. rhs 의 type이 포인터 여야 함. (값을 의미 X)
   - Pointer 타입을 Non-Pointer 타입으로 Cast 하여 진행하는 경우 추적 불가능. 예) uint64_t a = (uint64_t)ptr; (⇒Limitation)

Runtime Library - Goal

- trace()로 수집한 Object Relationship기반으로 dangling pointer의 nullify를 진행

Runtime Library - Shadow Object Tree

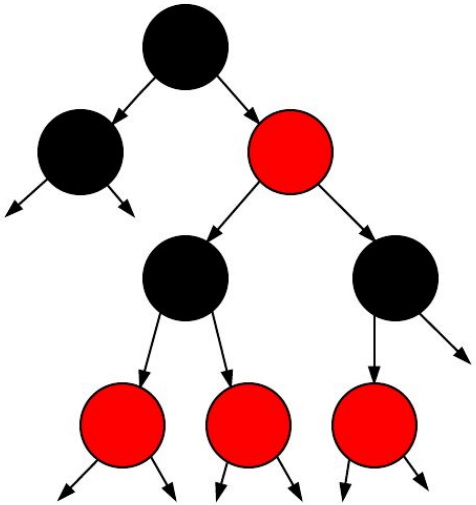
- 할당 시, 해당 메모리 오브젝트에 대한 shadowObject 생성 후, Tree에 삽입.



Runtime Library – Shadow Object Tree

- shadowObjTree 는 RBTree로 이루어져있음.

shadowObjTree



Performance 측면에서 Insertion, remove 보다 검색 Operation 많아도 괜찮다.

* trace(p) 호출마다 $(Object\ base <= given\ p < Object\ end)$ 를 만족하는 Object 를 찾아야 하므로 대체적으로 검색 횟수가 삽입/삭제 횟수 보다 많음.

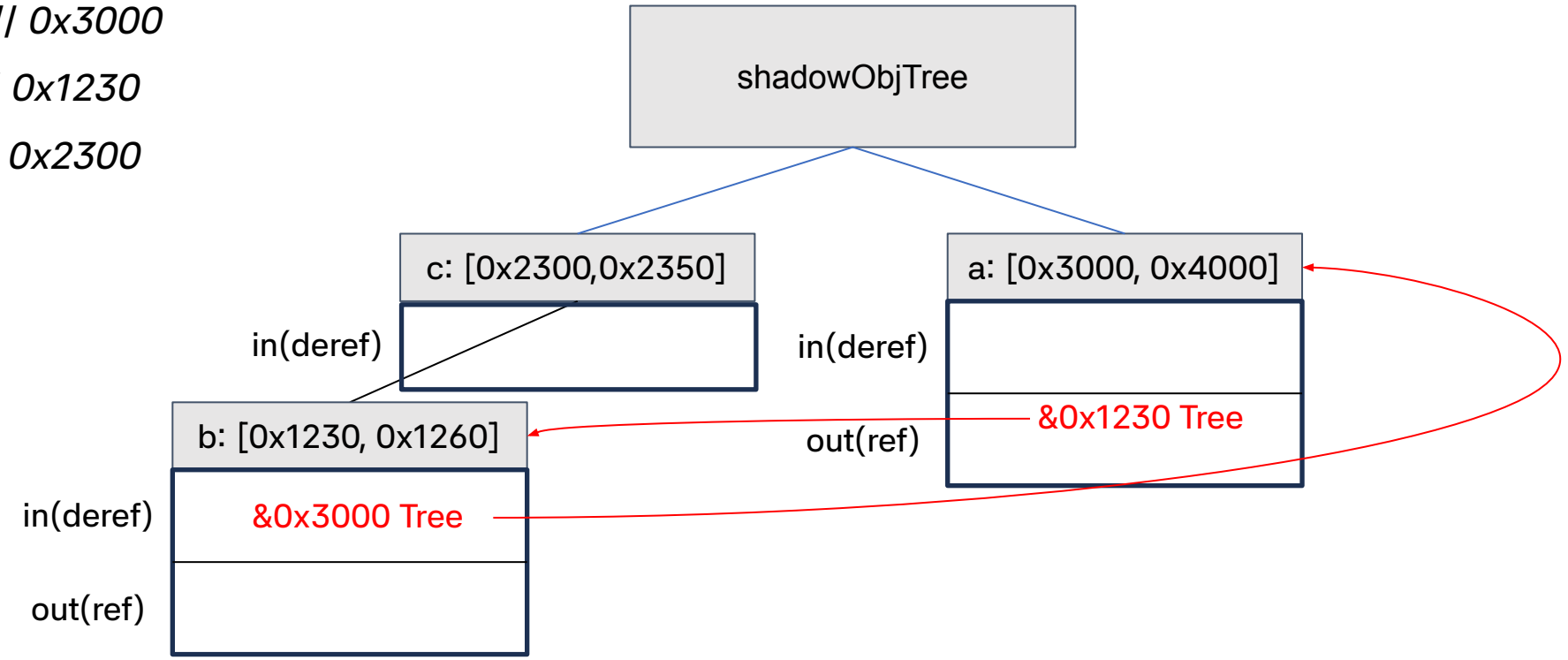
Runtime Library - SubTree

- shadowObjTree의 element에는 dereference(in) / reference(out) 를 담고 있는 subtree 있음.

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

*a = b;

*b = c;



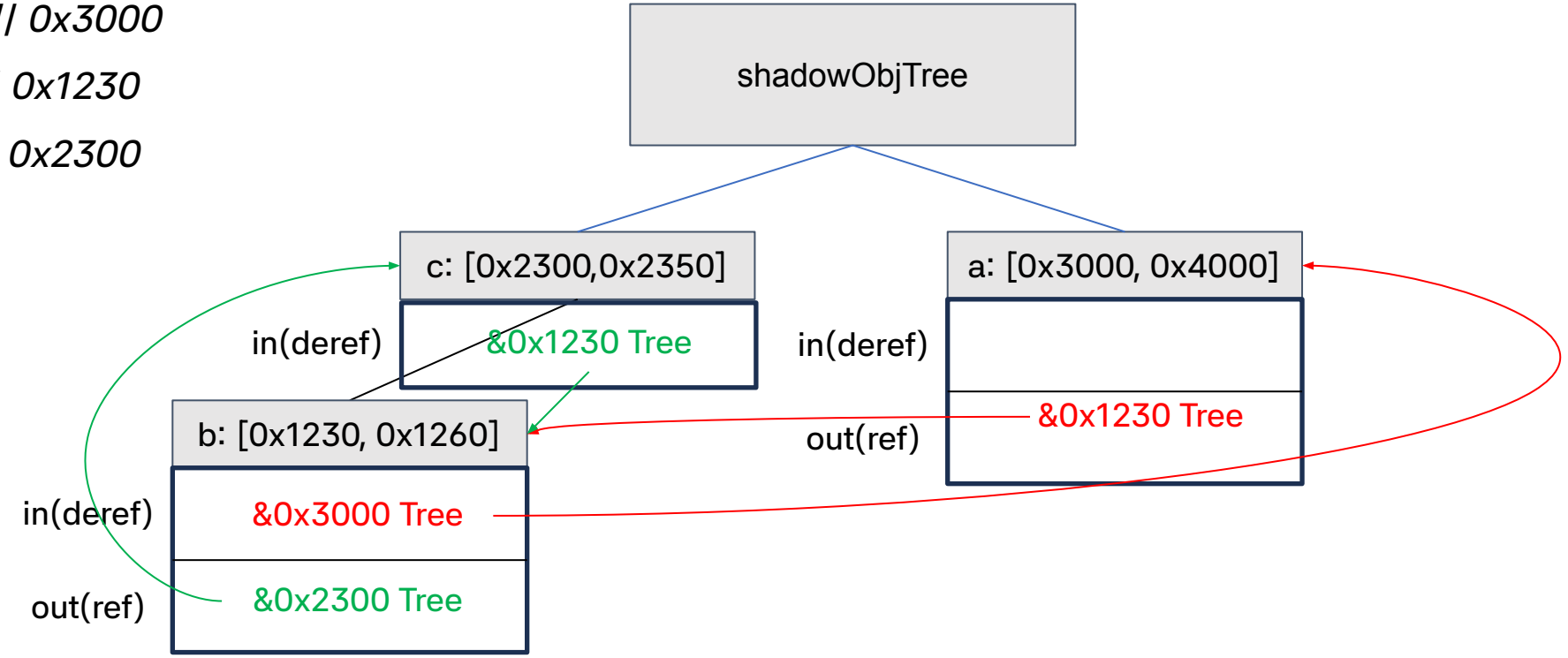
Runtime Library - SubTree

- shadowObjTree의 element에는 dereference(in) / reference(out) 를 담고 있는 subtree 있음.

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

*a = b;

*b = c;



Runtime Library - Runtime Operations and Nullification

```
1  def allocObj(size):
2     ptr = real_alloc(size)
3     shadowObj = createShadowObj(ptr, size)
4     shadowObjTree.insert(shadowObj)
5     return ptr
6
7  # NOTE. lhs <- rhs
8  def trace(lhs, rhs):
9     lhsShadowObj = shadowObjTree.find(lhs)
10    rhsShadowObj = shadowObjTree.find(rhs)
11
12    # Check if lhs and rhs are eligible targets.
13    if lhsShadowObj and rhsShadowObj:
14       removeOldShadowPtr(lhs, rhs)
15       ptr = createShadowPtr(lhs, rhs)
16       lhsShadowObj.insertOutboundPtr(ptr)
17       rhsShadowObj.insertInboundPtr(ptr)
18    return
19
20 def freeObj(ptr):
21    shadowObj = shadowObjTree.find(ptr)
22
23    for ptr in shadowObj.getInboundPtrs():
24       srcShadowObj = shadowObjTree.find(ptr)
25       srcShadowObj.removeOutboundPtr(ptr)
26       if shadowObj.base <= ptr < shadowObj.end:
27          *ptr = NULLIFY_VALUE
28
29    for ptr in shadowObj.getOutboundPtrs():
30       dstShadowObj = shadowObjTree.find(ptr)
31       dstShadowObj.removeInboundPtr(ptr)
32
33    shadowObjTree.remove(shadowObj)
34
35    return real_free(ptr)
```

- 할당/해제 함수 => allocObj / freeObj 로 교체
  - allocObj: 실제 할당 후, 해당 객체에 대한 shadowObject 생성하여 Tree 삽입
  - freeObj: ShadowObjTree 에서 free target object 를 찾은 후,
    이를 참조하는 포인터들의 nullify 진행 (자세한 내용은, 다음 장 참고)

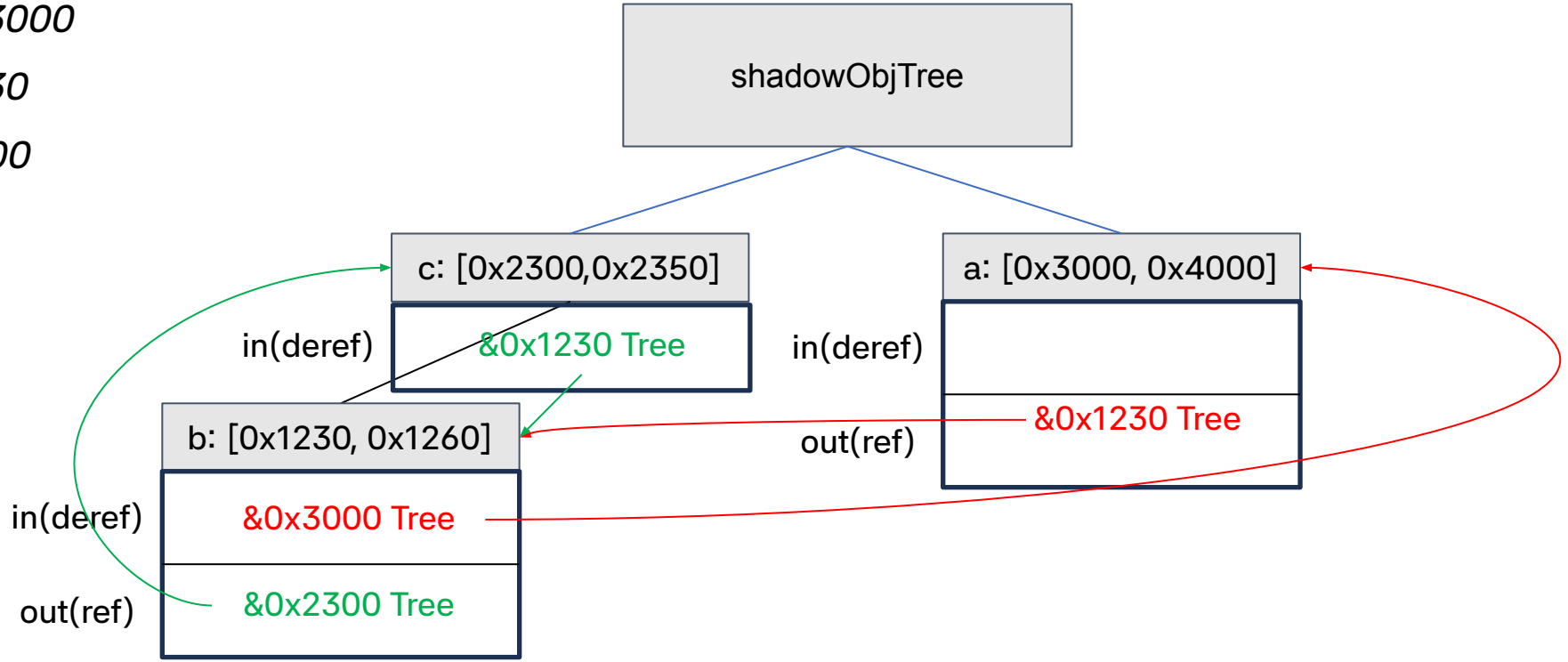
- trace: lhs, rhs에 대한 in / out 각각 추가

Runtime Library - Runtime Operations and Nullification

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

*a = b;

*b = c;



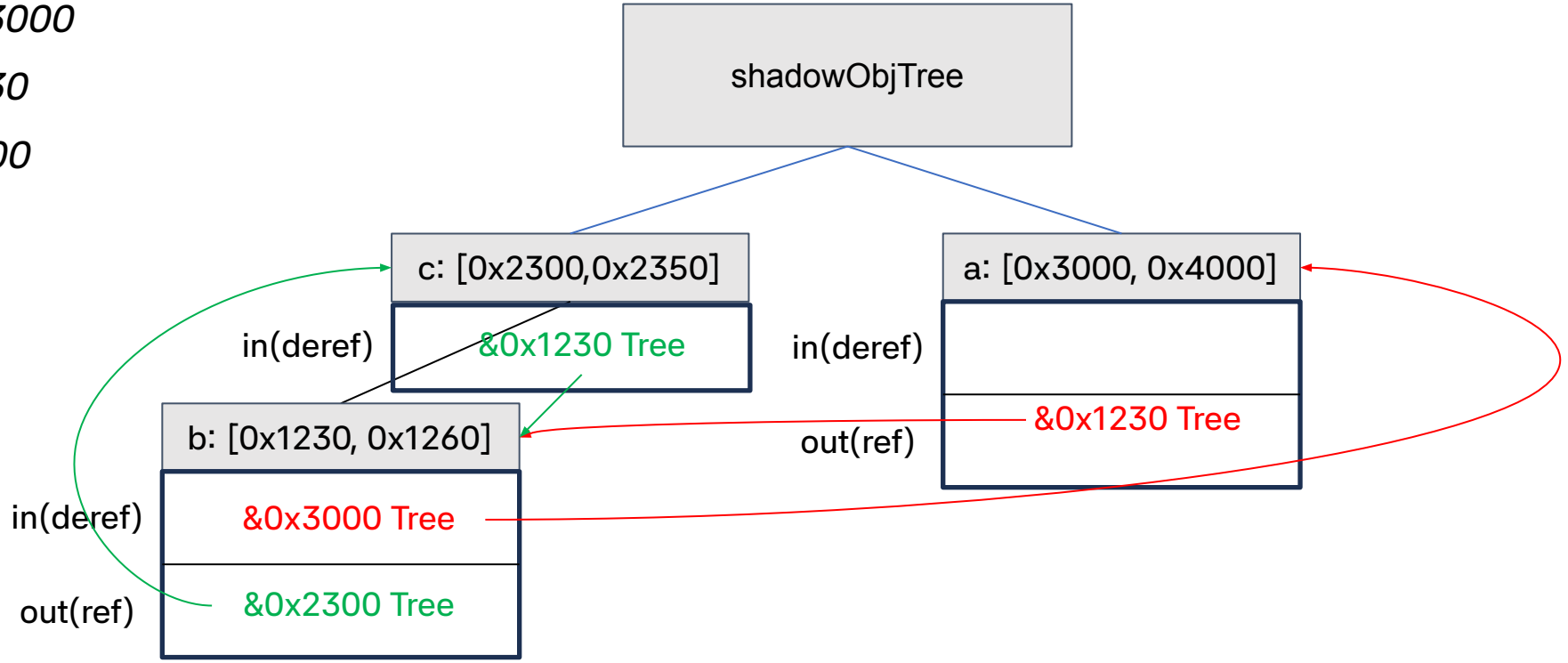
Runtime Library - Runtime Operations and Nullification

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

*a = b;

*b = c;

free(a);



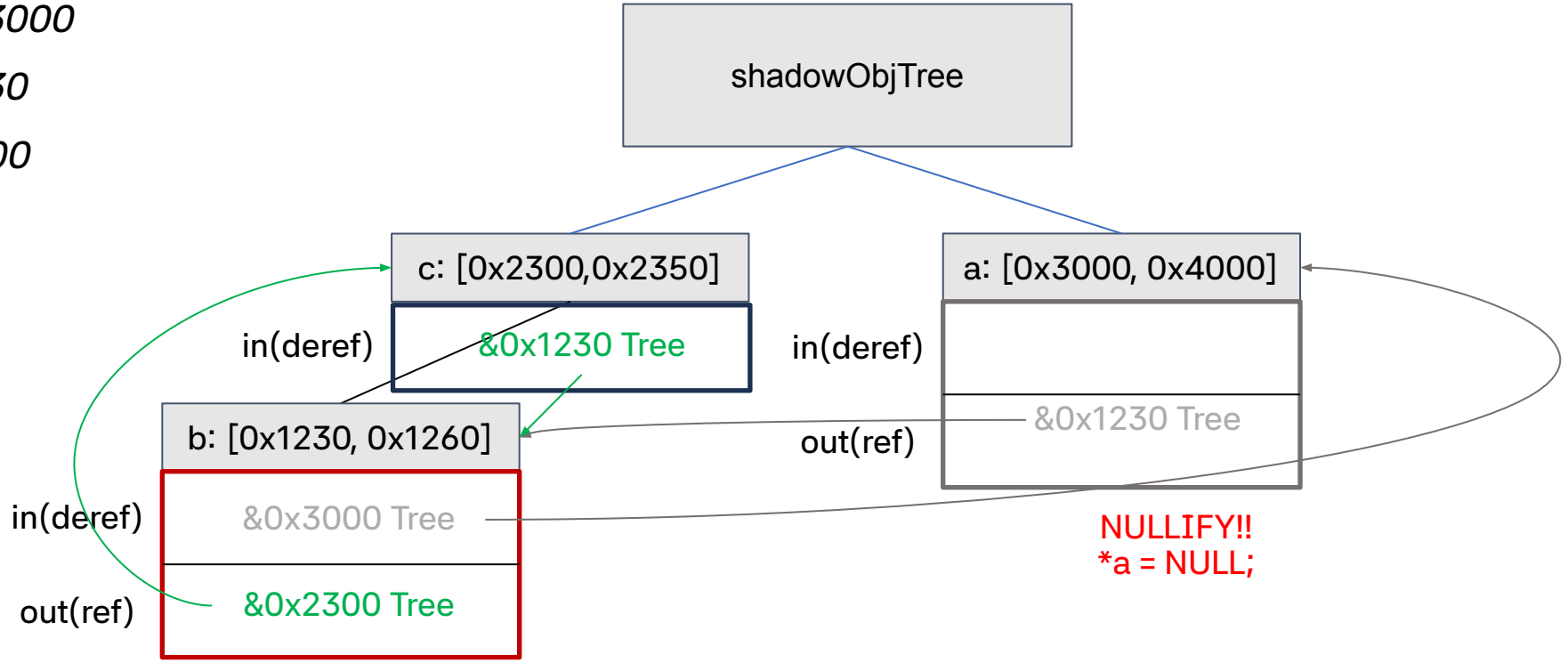
Runtime Library - Runtime Operations and Nullification

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

*a = b;

*b = c;

free(a);

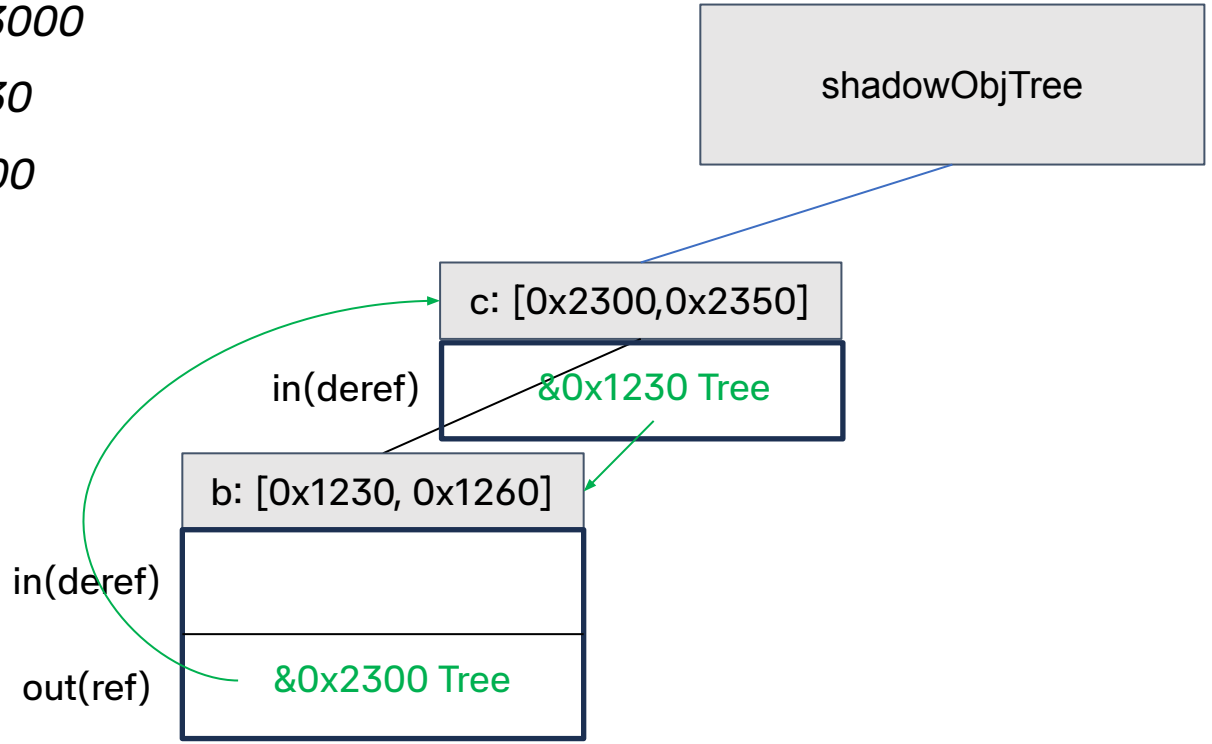


Runtime Library - Runtime Operations and Nullification

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

```
*a = b;
*b = c;

free(a);
free(b);
```

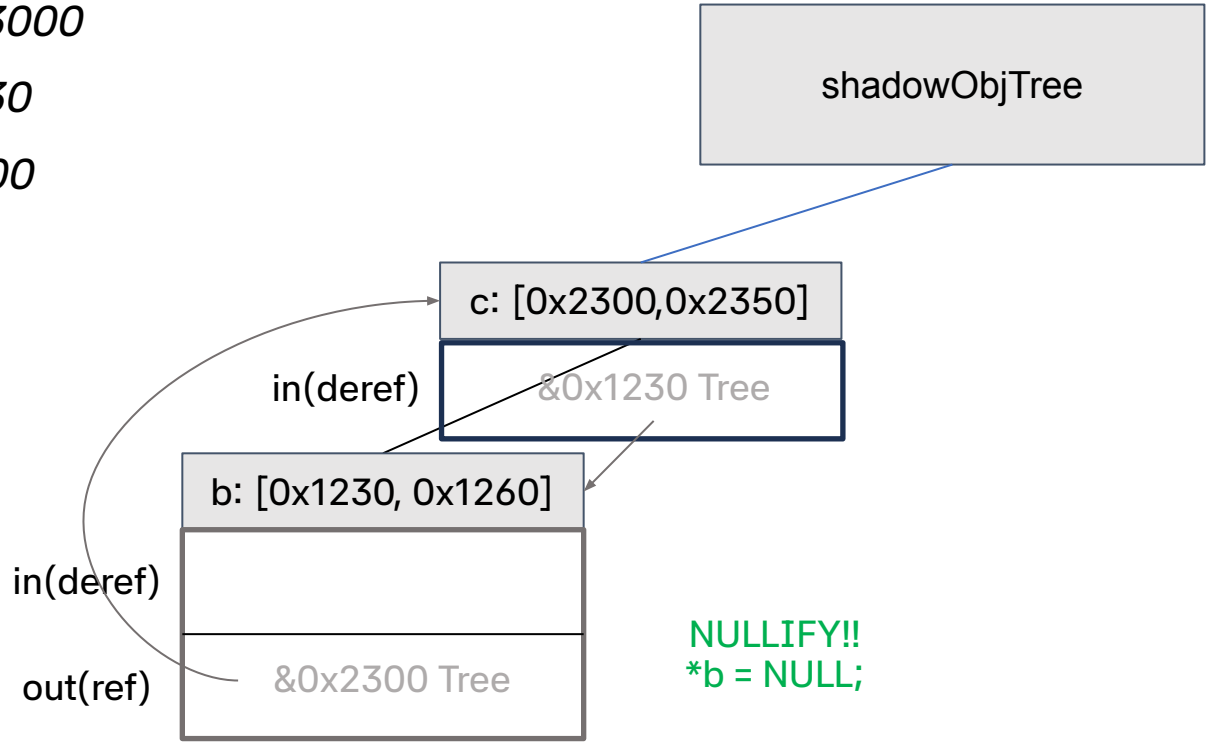


Runtime Library - Runtime Operations and Nullification

```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

```
*a = b;
*b = c;

free(a);
free(b);
```



Runtime Library - Runtime Operations and Nullification

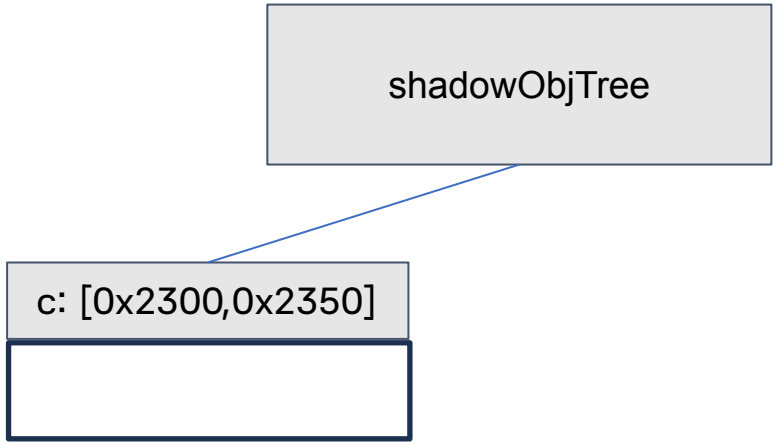
```
int **a = malloc(0x1000); // 0x3000
int **b = malloc(0x30); // 0x1230
int **c = malloc(0x50); // 0x2300
```

```
*a = b;
*b = c;
```

```
free(a);
free(b);
```

RESULT

a = = b == NULL



Runtime Library - Runtime Operations and Nullification

- Nullify를 함으로써, 그 후 생기는 dereference는 Null-Pointer 접근으로 SEGFAULT 를 발생시킴

Evaluation

- Environment

  CPU : quadcore 3.40 GHz CPU (Intel Xeon E3-1245)

  memory :16GB RAM

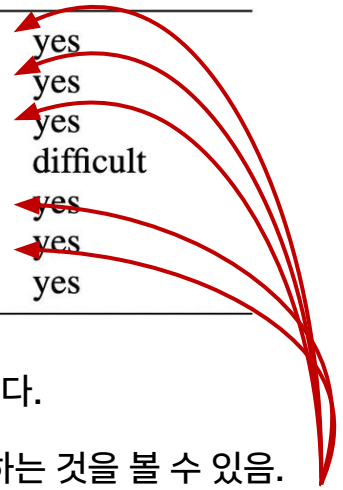
  OS : Ubuntu 13.10 system (Linux Kernel 3.11.0)

- Running-Time, Memory-Overhead

Attack Mitigation

- Chromium Browser – 7 CVEs

Bug ID	CVE	Severity	No-Nullify	Nullify-value				Control b/w free and use
				0	1	2	3	
261836	-	High	0x7f27000001a8	0x2e8	0x21	0x22	0x23	yes
265838	2013-2909	High	0x1bfc9901ece1	✓	0x1	0x2	0x3	yes
279277	2013-2909	High	0x7f2f57260968	✓	0x1	0x2	0x3	yes
282088	2013-2918	High	0x490341400000	0xf0	0xf0	0xf0	0xf0	difficult
286975	2013-2922	High	0x60b000006da4	✓	0x15	0x16	0x17	yes
295010	2013-6625	High	0x897ccce6951	0x30	0x1	0x2	0x3	yes
162835	2012-5137	High	0x612000046c18	*stopped by assertion*				yes



- DANGNULL를 적용 안 했을 때, 6개의 SEGFAULT 주소들은 Exploitable 했으므로 위험한 주소이다.
- 하지만, DANGNULL를 적용하고 Nullify를 0,1,2,3 으로 했을 때, 값이 이에 맞게 linear하게 증가하는 것을 볼 수 있음.

  ⇒ DANGNULL의 Nullify가 취약점 벡터에 효과적으로 영향을 미치고 있음을 확인.
- mark (✓) 는 DANGNULL를 적용했을 때, CHROME 자체의 pointer validation에 걸린 것이다. (= Expected Error/Behaviour)

  = > 즉, DANGNULL를 기존 pointer assertion을 re-utilize 시키는 역할도 함.

Instrumentation

- SPEC CPU 2006

Name	Lan.	File Size (KB) before	after	incr.	# of instructions inserted	total	# of objects total	peak	# of pointers total	peak	# Nullify	Memory (MB) before	after
bzip2	C	172	549	378	13	15,370	7	2	0	0	0	34	34
gcc	C	8,380	9,148	768	9,264	606,925	165k	3k	3167k	178k	104k	316	397
mcf	C	53	429	376	95	2,277	2	1	0	0	0	569	570
milc	C	351	737	386	71	24,024	38	33	0	0	0	2,496	2,500
namd	C++	1,182	1,56[illegible]	382	45	77,434	964	953	0	0	0	44	114
gobmk	C	5,594	[illegible]	[illegible]	201	156,829	12k	47	0	0	0	23	28
soplex	C++	4,292	[illegible]	453	264	74,314	1k	88	14k	172	140	7	14
povray	C++	3,38[illegible]	[illegible]	513	941	194,821	15k	9k	7923k	26k	6k	38	81
hmmer	C	814	,210	396	94	60,832	84k	28	0	0	0	1	18
sjeng	C	276	662	386	17	22,836	1	1	0	0	0	171	171
libquantum	C	106	483	378	21	7,301	49	2	0	0	0	0	2
h264ref	C	1,225	1,646	420	154	115,575	9k	7k	906	111	101	44	208
lbm	C	37	411	374	9	2,341	2	1	0	0	0	408	409
astar	C++	195	574	378	54	8,220	130k	5k	2k	148	20	13	135
sphinx3	C	541	931	389	170	34,476	6k	703	814k	14k	0	46	62
xalancbmk	C++	48,538	51,010	2472	7,364	645,434	28k	4k	256k	18k	10k	7	76

10배

DANGNULL에 의해 프로그램에 삽입된 instruction 개수는 프로그램의 전체 instructions 개수에도 영향 받지만,

"어떻게(할당, pointer assignments)" 구현되어 있는 지에 대해서도 영향 받는다.

예) mcf 와 lbm 을 비교하면 mcf 가 instructions 개수는 작지만 삽입된 instructions의 개수는 10 배 넘는다.

Instrumentation

- Chromium

Name	File Size (MB)			# of instructions	
	before	after	incr.	inserted	total
Chromium	1858	1868	10	140k	16,831k

Table VI: Static instrumentation results on Chromium

Chromium의 경우, 프로그램 크기의 1% 정도 instructions가 삽입 됐으며 파일 크기는 0.5% 늘어났다.

Runtime Overheads

- SPEC CPU 2006

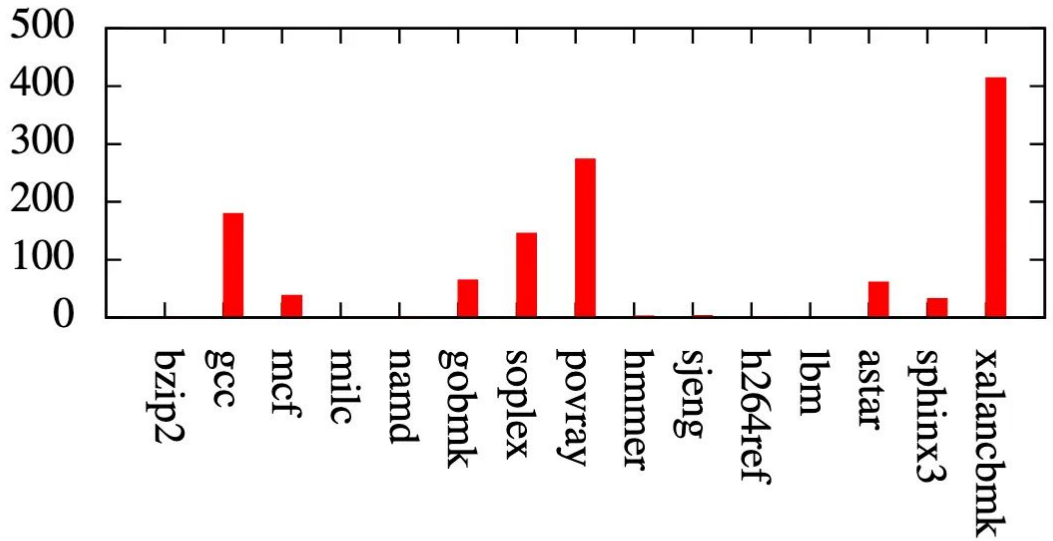


Figure 3: Runtime speed overheads when running SPEC CPU2006. y-axis shows slowdowns (%) of DANGNULL.

평균적으로는 80% Overhead 증가

povray 같은 경우는 포인터의 개수가 많아
실행시간 270% 증가,
메모리 오버헤드도 213% 증가

포인터 개수가 작은 h264ref 은
1% 실행시간 증가, 472% 메모리 오버헤드 증가

Runtime Overheads

- Several Browsers

Benchmarks (unit, [high or low])	JavaScript			Rendering			
	Octane (score, high)	SunSpider (ms, low)	Dromaeo JS (runs/s, high)	Balls (fps, high)	Dromaeo DOM (runs/s, high)	Dromaeo JS Lib (runs/s, high)	html5 (sec, low)
Original	13,874	320.0	1,602.1	11.6	857.8	216.0	10.1
DANGNULL	13,431	347.5	1,559.6	6.5	509.1	168.1	20.7
Slowdown	3.2%	8.6%	2.7%	44.1%	40.7%	22.2%	105.3%

3개의 Javascript 엔진에서는 실행시간 평균 4.8% 증가

대부분 JIT으로 컴파일 되기 때문에, 해당 부분은 DANGNULL의 범위를 벗어나서 별로 안늘어났다.

(DANGNULL은 Static analysis에서 instructions를 분석하는데 ,

Dynamic으로 생성되는 JIT은 DANGNULL의 범위 밖, LIMITATION)

렌더링은 53.1% 정도 렌더링 computations가 증가했다.

Compatibility

- False-Positive

- Program의 semantics를 바꿈으로써 생길 수 있는 False Positive를 측정하기 위해, Chrome으로 stress tests를 했지만 문제 없었다.
- 정말 정말 rare한 케이스가 있지만, 30, 000개 테스트한 결과 문제가 없음을 확인

Limitation

- Performance 부분에 있어서 개선이 필요하다.
- 실제 사용되지 않는 benign dangling pointer에 대해서도 Nullify 수행하여 이론적으로는 False-Positive 가능성이 존재함.
- 포인터 타입만 추적해서, 포인터 타입이 아닌 변수를 통한 propagation은 못 잡음.
- Heap에 담겨 있는 포인터 변수만 추적 가능.
- Dynamic으로 생성되는 instructions (ex. JIT)에 대해서는 못 잡음.