引言

内存错误的危害

缓冲区溢出和使用后释放（UAF）是导致生产环境中安全问题的主要原因。

空间错误：访问超出对象边界的内存（缓冲区溢出）

时间错误：访问超出对象生命周期的内存（使用后释放）

现有检测工具的局限性

AddressSanitizer (ASan)基于"红区"和逐地址检查的机制导致显著运行时开销 ASan需要迭代扫描整个内存范围以查找红区，而非一次性检查整个内存范围基于指针的检测与指针上的整数算术不兼容

RSan解决方案的必要性

RSan是一种新型的基于红区的内存错误检测器

引入创新的元数据和检查范式，结合红区的兼容性与丰富的每对象元数据格式将边界信息存储在每个内存对象关联的下溢红区中

通过指针标记和2的幂次大小类，能够快速定位元数据并一次性验证对任意内存范围的访问

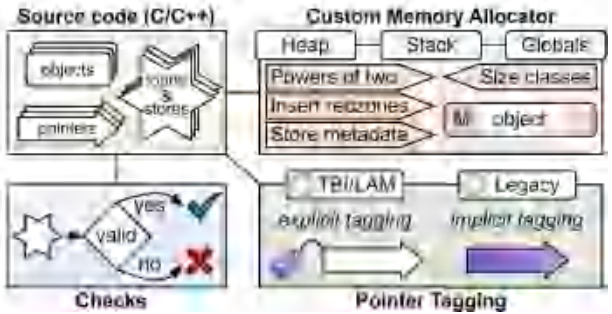
例1：在释放内存后使用该内存

```
int* p = (int*)malloc(sizeof(int));
free(p);
*p = 1;                    // use-after-free
free(p);                   // double-free
```

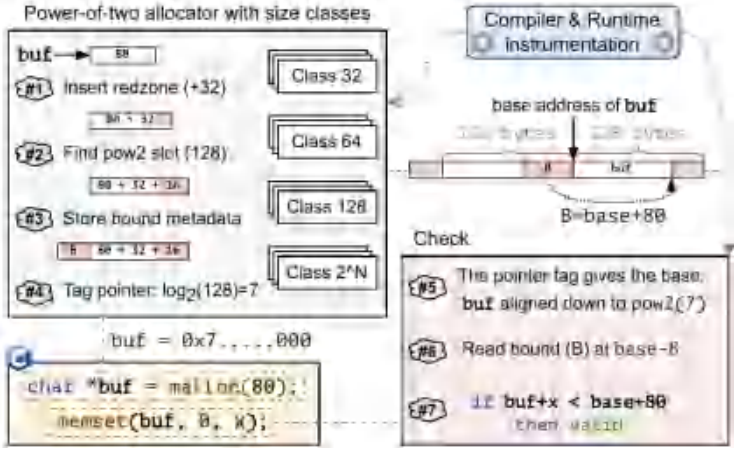
例2：越界读取或者写入

```
void foo() {
    char buffer[8];
    strcpy(buffer, "AAAAABBBBB"); // buffer-overflow
}
```

RSan 概述



RSan组件概述



RSan检测流程

RSan 设计——分配器

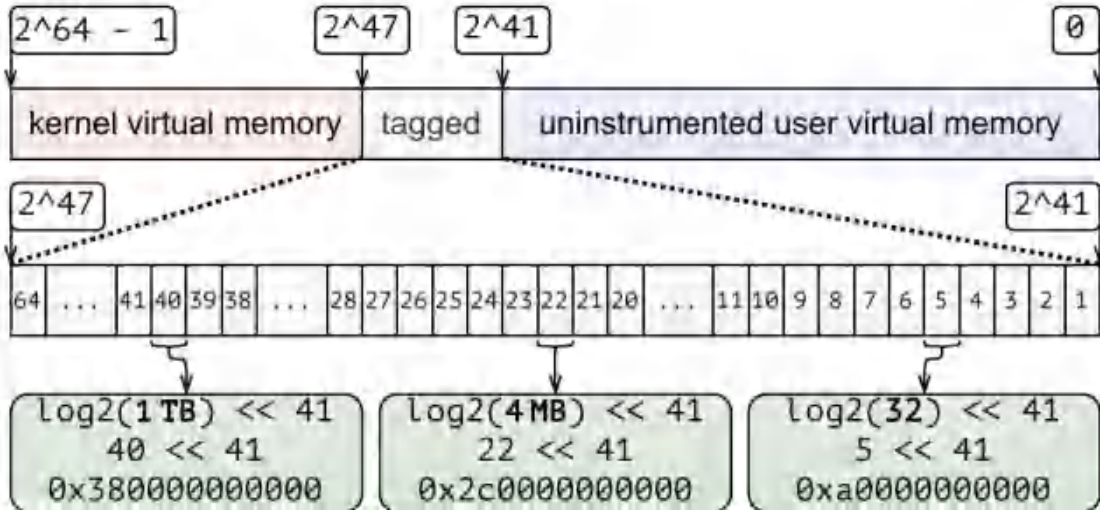
- 常见的内存检测器需要维护元数据，比如对象的基址和大小。
- 如果把这些信息单独存到一个map/shadow memory里，访问时要查一一开销大

RSan 不想额外查 map，而是利用**指针的高位（未用的比特位）** 来存元数据，也就是**pointer tagging**。使用定制内存分配器让其始终为2的幂次方大小，

对于堆，诸如TCMalloc、Scudo和jemalloc等流行的分配器已经提供了大小类，并且可以很容易地限制为2的幂的类。
对于栈和全局内存，将栈被分成多个栈，每个栈专门用于特定2的幂大小类的对象。全局变量被移动到全局数组中，每个数组包含特定大小类的条目。

RSan 设计——指针标记

显示标记：Arm和Intel CPU的TBI和LAM特性（存放在地址的高位当中）

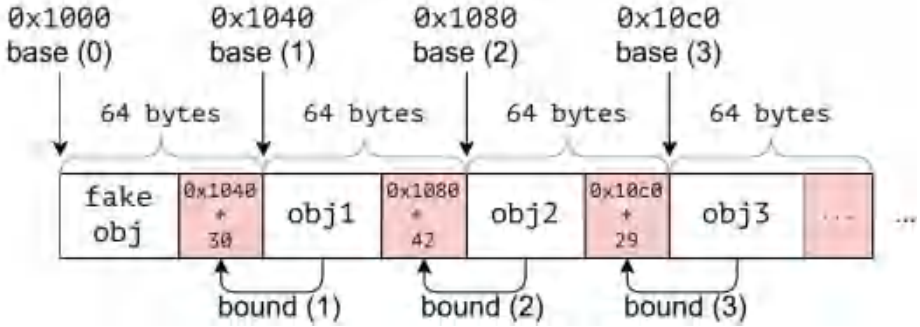


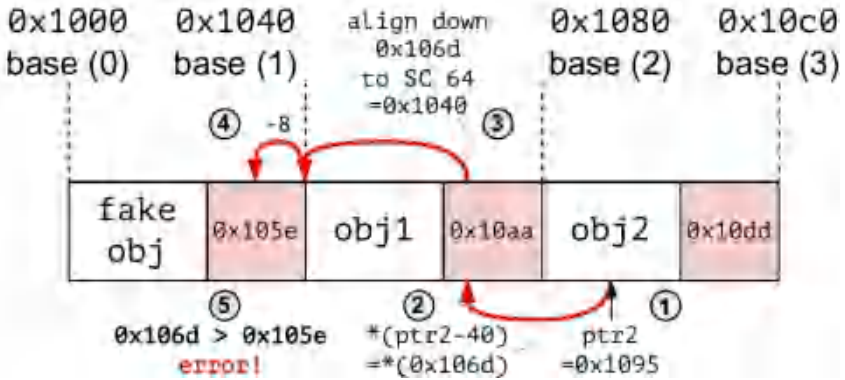
隐式标记

一共 6 位 → 能表示 2^6 =64 个不同值。
地址空间必须被切成 64 份，每份对应一个 tag 值。
所以每块大小**刚好 2^47/2^6=2 TB**

RSan 设计——元数据

RSan将当前对象的元数据（边界信息）存储在前一个对象槽的redzone填充中。





RSan 优化

现有优化

- 移除不可满足的检查
- 移除重复检查
- 优化相邻检查

循环优化

无条件执行的访问：

- 循环不变指针检查移出循环
- 使用SCEV分析循环变量指针范围

有条件执行的访问：

- 缓存元数据查找结果
- 对只增加的循环变量指针缓存元数据

范围检查合并

- 常量偏移合并
- 负-正合并
- 低-高合并

优化对运行时开销的影响

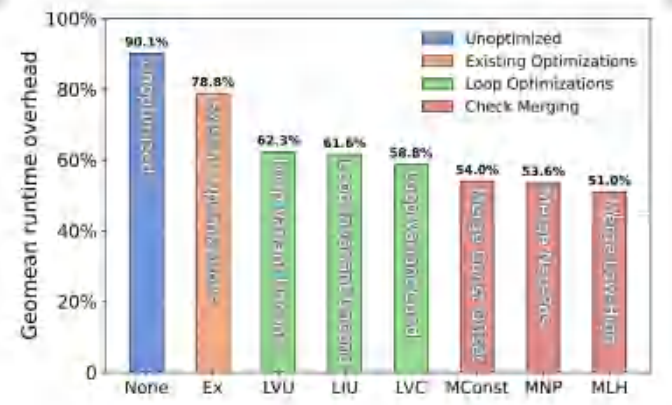


Figure 8: Runtime overhead progression of RSan on SPEC CPU2006 when enabling optimizations one-by-one.

未优化的RSan开销为90.1%，经过全部优化后降至51%

RSan 优化——循环优化

无条件的访问

- 循环外提
  - 会和UAF产生冲突
  - 循环前:假阴性（漏报）
- 范围检查
  - SCEV求出地址始末地址
  - 循环后:假阳性（误报）

ASan--通过避免在循环包含以目标指针为参数的调用时进行优化来解决这个问题（这表明可能存在释放）。GiantSan包含一个循环后的use-after-free检查，如果循环中没有发生use-after-free，则会出现假阳性。

Listing 4 Hoisting a loop invariant check.

```
check(&ptr[x])
for(uint i = 0; i < n; i++)
  ptr[x] = 0; // ptr[x] is loop invariant
```

Listing 5 Hoisting a loop variant *range* check.

```
range_check(&ptr[s], &ptr[e])
for(uint i = s; i <= e; i++)
  ptr[i] = 0; // ptr[i] is loop variant
```

RSan 优化——循环优化

Listing 6 Caching the bound metadata in a local variable for loop variant conditional accesses. `check_ret_meta()` is a regular check that returns the found metadata for reuse.

```
uint64_t bound = 0;
for(uint i = 0; i < n; i++){
  if( cond ){ // conditional
    if( &ptr[i]+access_size > bound ){
      bound = check_ret_meta(&ptr[i]);
    }
    ptr[i] = 0; // ptr[i] is loop variant
  }
}
```

只考虑单调递增

RSan 优化——检查合并

Listing 7 Merging the check for a constant offset range.

```
range_check(&ptr[-10], &ptr[80]);
ptr[-10] = 0; ptr[15] = 0; ptr[80] = 0;
```

Listing 8 Merging the check for a negative and positive pair.

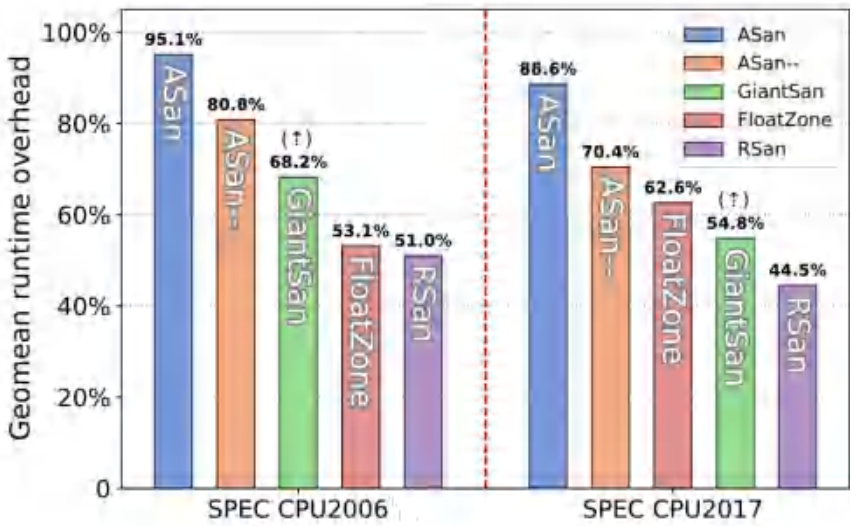
```
int neg = -x; uint pos = x; // proven neg/pos
ptr[neg];                   // load
range_check(&ptr[neg], &ptr[pos]);
ptr[pos] = 0;               // store
```

Listing 9 Merging the check for two pointers to the same object by computing the lower and higher address.

```
int x = ..., y = ...; // both variable
ptr[x];               // load
if( &ptr+x > &ptr+y )
  range_check(&ptr[y], &ptr[x]);
else
  range_check(&ptr[x], &ptr[y]);
ptr[y] = 0;           // store
```

实验结果

Description (CWE)	Total	ASan	RSan
Stack buffer overflow (121)	2,885	2,791	2,885
Heap buffer overflow (122)	3,365	3,318	3,365
Buffer underwrite (124)	1,001	907	1,001
Buffer overread (126)	657	563	657
Buffer underread (127)	1,001	907	1,001
Double free (415)	799	799	799
Use-after-free (416)	374	374	374



RSan在检测错误的时候没有产生漏报以及运行时开销均优于现有的工具。