

基于显式锁钥方案的 UAF 漏洞检测方法的研究和改进

摘要

释放后使用 (use after free, UAF) 漏洞严重危害软件安全。相比起其他内存漏洞, UAF 漏洞更难通过人工排查或静态分析发现。基于证据的动态检测工具一般只记录指针或对象的合法性, 但是悬空指针指向的内存被再次分配后, 现有工具出现不同程度的漏报。为了解决这一问题, UAFSan 设计了显式锁钥方案, 使用唯一标识符单独标识堆对象, 可以在内存检查时通过指针和堆对象的标识符的一致性判断再分配是否发生。我们发现并修正了动态 UAF 检测工具 UAFSan 的源代码的 2 个错误, 并发现它存在漏报使用标准 C 库函数 realloc() 的 C/C++ 程序的 UAF 漏洞的情况, 我们给出 2 种改进方案: 1) 改进重分配管理算法; 2) 改进内存访问检查算法。实验结果证明基于改进方案实现的 UAFSan 提高了检测 UAF 漏洞的准确性, 且产生的时间和内存开销可以忽略。

关键词: 释放后使用 (use after free, UAF); 双重释放 (double free, DF); 插桩; 动态分析; 锁钥方案

1 引言

当 C/C++ 程序的内存对象释放后, 指向该对象的指针成为悬空指针。使用悬空指针访问已经释放的内存对象, 会产生释放后使用 (use after free, UAF) 漏洞。以图1程序为例, 第 4 行代码申请分配堆对象 obj1, 其首地址返回至指针 p, 第 6 行代码中, 指针 q 也保存了该地址。第 7 行代码将 obj1 释放后, p 和 q 均为悬空指针。第 10 行代码使用悬空指针 q 访问 obj1, 产生 UAF 漏洞; 第 11 行释放悬空指针 q 的内存对象, 产生双重释放 (double free, DF) 漏洞。本文仅讨论堆内存上的 UAF 漏洞, 且将 DF 视为 UAF 的特殊形式。

UAF 漏洞对软件的安全和鲁棒性带来严重危害, 原因如下: 1) UAF 漏洞可能会造成程序崩溃、静态数据泄漏和遭到任意代码执行攻击等严重问题 [21,25]; 2) 近年来攻击者利用 UAF 漏洞对程序运行环境实施攻击的行为与日俱增。有调研报告指出, UAF 漏洞已经成为 Chrome 中最普遍的安全漏洞之一, 攻击者可以利用 UAF 漏洞完全控制被入侵的计算机 [11,23]; 根据通用漏洞披露 (common vulnerabilities and exposures, CVE) 数据库¹, 从 2010 年到 2022 年间, UAF 漏洞的数量一直在快速增加且保持在较高水平 [13]。因此, 研究 UAF 漏洞的检测方法, 对提高互联网软件的安全和鲁棒性有举足轻重的意义。

¹<https://cve.mitre.org/index.html>

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p = (int*)malloc(40); // 分配obj1
5     printf("addr 1: %p\n", p);
6     int *q = p;
7     free(p); // 释放obj1
8     int *r = (int*)malloc(40); // 分配obj2
9     printf("addr 2: %p\n", r);
10    *q = 1; // use-after-free
11    free(q); // double-free
12    *r = 1;
13 }

```

图 1. 包含 UAF 漏洞的 C 程序

2 相关工作

现代软件的代码量不断增大，UAF 漏洞的排查日趋复杂，人工排查的难度日益增加，学术界和工业界提出了多种自动化检测的方法，主要分为动态 UAF 漏洞检测和静态 UAF 漏洞分析 2 类。

2.1 动态 UAF 漏洞检测

动态 UAF 漏洞检测在程序运行时采集程序的行为，并据此检测是否存在 UAF 漏洞。主要分为基于证据（evidence-based）的检测方法和基于预测（prediction-based）的检测方法 2 种。

基于证据的检测方法跟踪指针或对象，以记录指针的合法性或对象的状态。DoubleTake [12] 使用库插入器来动态检测程序，并且只能检测写入已释放内存的 UAF 漏洞。ASan [18] 在编译时检测程序，因此产生较低的运行时开销。valgrind [16]、QASan [6] 和 Dr.Memory [1] 使用动态二进制插桩，因此引入大量的运行时开销。EffectiveSan [4] 使用胖指针（low-fat pointer）来记录指针可以安全访问的内存地址，但因改变了内存布局而引入兼容性问题。为避免兼容性问题，CETS [15]、MemSafe [22] 和 MOVEC [2] 使用采集元数据的方法跟踪指针。TSan [19,20] 旨在检测数据竞争，因此可能无法检测与数据竞争无关的 UAF 漏洞。

基于证据的检测方法检测顺序程序的 UAF 漏洞较为准确，但是存在痛点问题：如果已经释放的内存被再分配，悬空指针指向的内存区域是有效的，现有方法将会不同程度地漏报 UAF 漏洞。图1程序在运行时可能出现 2 种现象，如果内存再分配未发生，如图2 a) 所示，第 8 行代码将堆对象 obj2 分配在和 obj1 不同的位置；如果内存再分配发生，如图2 b) 所示，第 8 行代码将堆对象 obj2 分配在和 obj1 相同的位置，尽管再次使用指针 q 访问内存 obj1 会产生 UAF 漏洞，但由于该地址上 obj2 的存在，指针 q 指向的内存地址是合法的。在 Ubuntu 18.04 上的测试表明，图1程序总是进入图2 b) 的情况，第 11 行对象 obj2 释放后，指针 r 悬空，因此第 12 行 *r = 1 又引入 1 处 UAF 漏洞。

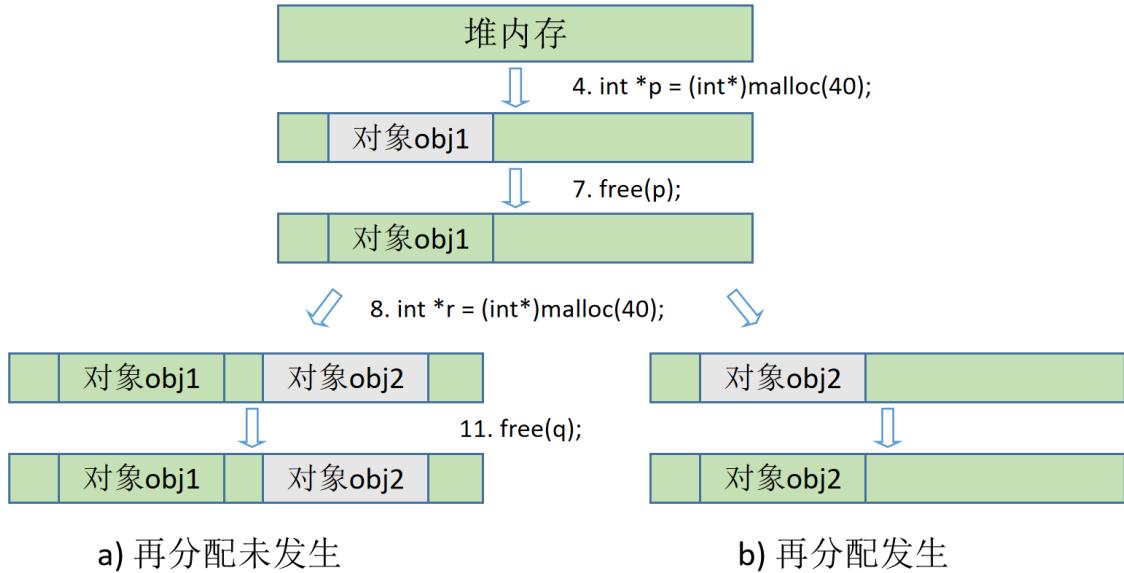


图 2. 图1程序可能出现的 2 种情况

内存的安全重用问题可以使用如下 2 种方法解决。

1) **延迟重用 (reuse delay)**。当堆内存释放后，我们并不马上将其还给操作系统，而是先保存在缓存队列中，等到缓存队列达到一定规模后再真正释放。valgrind [16]、ASan [18]、DoubleTake [12] 和 Dr. Memory [1] 均采用了这种策略，但是存在性能和准确性的折中：如果缓存队列太大，将导致大量的运行时内存开销；如果太小，则容易因内存再分配而漏报 UAF 漏洞。因此我们认为延迟重用技术并不能确保内存的安全重用，桂滨法等人的实验也证明了上述延迟重用的工作会漏报部分公开的 UAF 漏洞 [8]。

2) **锁钥方案 (key-lock scheme)**。显式锁钥方案在创建堆对象时，生成唯一标识符给对象上锁，给原始指针及其派生指针配发相同的标识符作为钥匙，通过钥匙和锁的一致性来判断内存重用是否发生。CETS [15] 和本文方法 UAFSan [8] 均属于此范畴，然而桂滨法等人的实验说明 CETS 的原型系统的准确性和鲁棒性较差，无法检测一些真实世界的大型软件漏洞。由于追踪指针传播需要较大的性能开销，UAFSan 在编译插桩阶段使用保守数据流分析缓解。隐式锁钥方案并不生成唯一标识符作为指针访问对象的锁和钥匙，而是使用系统或硬件的特性接管漏洞检测。Oscar [3] 和 Dangzero [7] 引入别名页 (alias page) 技术，利用硬件的内存管理单元 (memory management unit, MMU) 的特性实现内存安全重用，所不同的是 Oscar 仅缓解 UAF 攻击，Dangzero 同时拥有漏洞检测功能，并使用垃圾回收算法回收别名地址以防止耗尽。Safe Sulong [17] 将 C/C++ 程序编译为可以使用 Java 虚拟机运行的字节码，通过 Java 虚拟机抛出异常来保护系统安全并检测漏洞，并在其性能实验中表现出可观的前途。然而，将 C/C++ 的各种类型转化为 Java 类型的过程中，各种兼容性问题是不可避免的，因此 Safe Sulong 目前并不能用于真实世界的大型软件的检测。

基于预测的 UAF 检测方法旨在预测并发程序中的 UAF 漏洞，如 UFO [10] 和 ConVul [14]。他们不考虑非并发程序中的 UAF 漏洞。

2.2 静态 UAF 漏洞检测

静态分析是指对程序源代码或目标代码进行分析，发现程序漏洞并报告。和其他的内存漏洞不同，UAF 漏洞很难使用人工排查和静态分析的方法检测，原因如下：1) 我们很难追踪推断在复杂的数据结构之间所有的指针的别名；2) 我们很难确认程序在运行时指针的具体指向；3) 大型软件中复杂的路径使过程间分析（inter-procedural analysis）变得相当困难。现有的静态 UAF 漏洞分析的方法，由于使用有限的过程间分析和不完全的指针分析，可能误报或漏报 UAF 漏洞。GUEB [5] 使用专用值分析来跟踪堆操作和指针传播。但是，由于它的过程间分析是基于函数内联的，它可能会重复分析同一个函数。CRed [25] 基于需求驱动的指针分析和时空上下文缩减来检测 UAF。与 CRed 类似，Tac [24] 也是基于需求驱动的指针分析，使用机器学习来减少误报。

3 本文方法

3.1 本文方法概述

UAFSan 由编译插桩模块和运行库组成，如图3，编译插桩模块负责堆内存函数替换、指针采集、内存检查。运行库实现了插桩函数的细节，负责程序的堆对象管理、内存访问检查和指针追踪。使用 UAFSan 进行软件测试时，编译插桩模块将待测试程序的源代码进行编译成中间表示并插桩，再用 LLVM 编译器后端进一步编译并链接运行库，得到插桩程序。插桩程序包含了 UAFSan 的插桩代码，在没有漏洞时正常执行，检测到漏洞时报告漏洞。

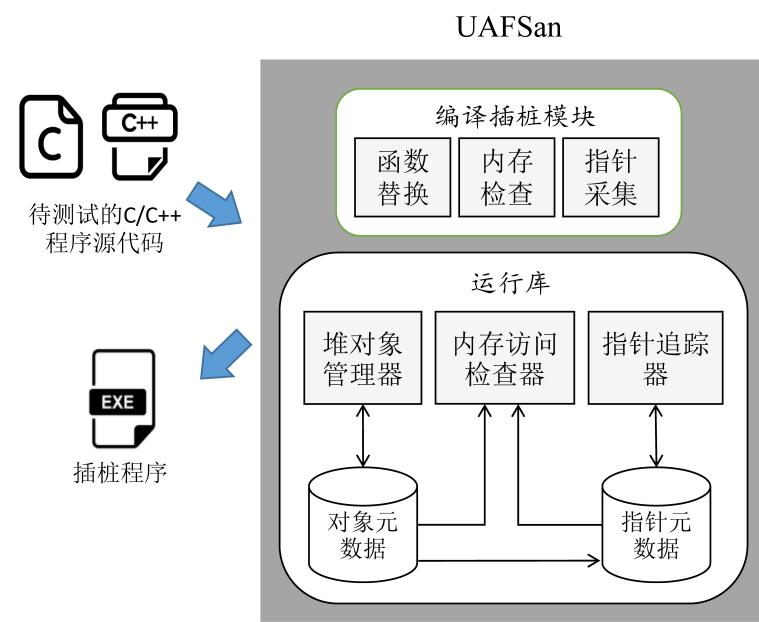


图 3. UAFSan 的设计框架

3.2 元数据投影表结构

UAFSan 检测 UAF 漏洞需要记录堆对象和指针的元数据，元数据结构 POmetadata 包含 3 个字段：对象标识符、对象首地址和对象大小，各占 8 字节。标识符用以区分对象之间的彼此不同。对象首地址用于检查该对象是否合法，非法时标记为 NULL。对象大小即对象

的字节数。指针被赋值时，该指针将得到其指向的堆对象的元数据。图4展示了图1程序运行完第8行后，指针 p 和 q 同时指向对象 obj1 的关系在元数据投影表上形成的3条记录。

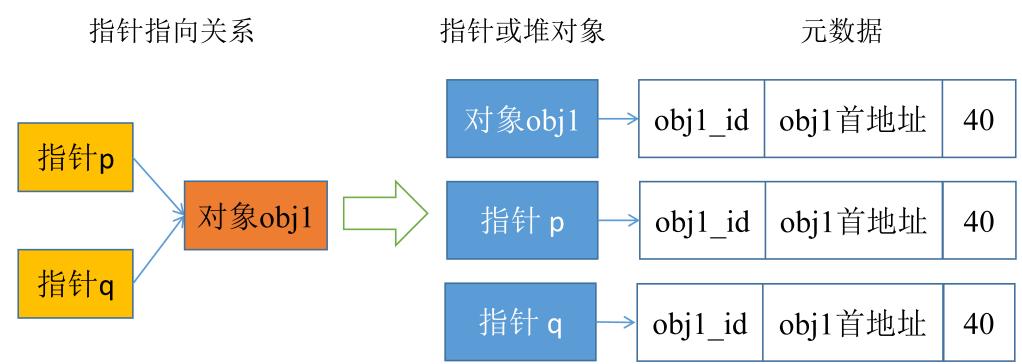


图 4. 指针指向堆对象的关系在元数据投影表上的记录

根据 64 位操作系统的虚拟进程空间布局，虚拟内存地址中高 16 位为内核地址，因此堆内存地址的高 16 位为 0；内存地址按 8 字节对齐，因此虚拟内存地址低 3 位也为 0。如图5所示，UAFSan 将中间 45 位的内存地址投影到二级表，其中高 23 位索引一级表，低 22 位索引二级表。

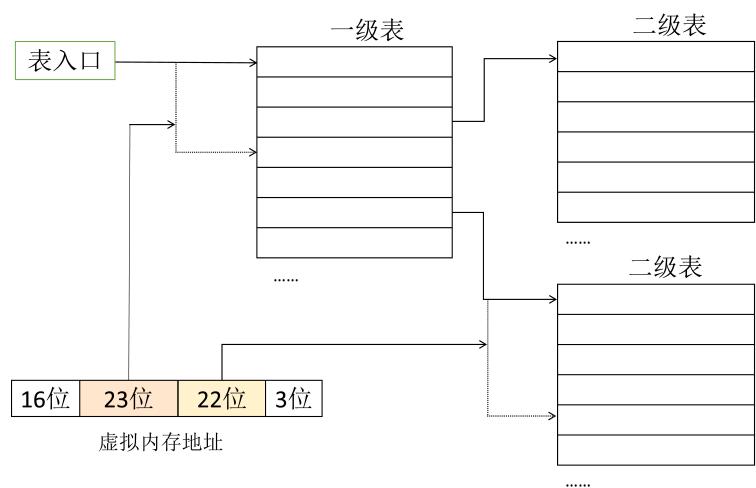


图 5. 元数据二级投影表

UAFSan 使用投影表存储指针元数据和堆对象元数据，维护指针或对象到元数据的映射关系。查找方法是传入指针的地址，找到指针元数据；或传入对象地址，找到对象元数据。为方便叙述 UAFSan 的实现细节，如图6所示，我们引入 UAFSan 元数据投影表的编程接口。

```

1 // 1. 通过指针的地址查找指针元数据。
2 P0metadata* P0pmd_tbl_lookup(ptr_addr)
3 // 2. 通过对象的地址查找对象元数据。
4 P0metadata* P0omd_tbl_lookup(obj_addr)
5 // 3. 初始化原始指针的元数据。
6 void P0pmd_tbl_init_as(&original_ptr, original_ptr)
7 // 4. 初始化或更新派生指针元数据。
8 void P0pmd_tbl_init_or_update_as(&derived_ptr, &original_ptr)
9 // 5. 初始化或更新对象元数据。
10 void P0omd_tbl_init_or_update_as(obj_addr, obj_id, obj_addr/NULL, size)
11 // 6. 记录某1个对象的调用栈。调用栈记录了函数跳转过程中每一级的返回地址,
12 // 记录某1堆对象分配或释放过程中经过的UAFSan的函数路径。
13 void P0rcs_tbl_update(obj_id, allocated/freed)
14 // 7. 通过指针进行内存访问检查。
15 P0check_mem_access(pointer, pointer_metadata)

```

图 6. 元数据投影表的编程接口

3.3 实现细节

3.3.1 函数替换

UAFSan 在程序操作堆内存时跟踪并记录其行为。需要关注的有 C 语言的函数 malloc()、free()、realloc()、calloc()、mmap()、munmap()，和 C++ 的运算符 new()、new[]()、delete、delete[]() 等。编译插桩模块用包裹函数替换原有的堆内存操作函数和运算符。图1程序的 malloc() 和 free() 被替换为图7的 wrapped_malloc() 和 wrapped_free()，包裹函数调用原生的 malloc() 和 free() 确保程序运行，同时加入追踪和检查代码。

3.3.2 指针采集

在 C/C++ 程序中，指针变量的赋值行为主要分为如下 2 类：1) 函数内的指针赋值；2) 函数间的指针传播。指针的初始化和赋值、指针在函数间的传播发生时，UAFSan 使用插桩函数更新指针元数据。

当 1 个指针被赋值时，我们初始化或更新指针元数据，分为以下几个步骤。首先，查找所有的形如 “ $p = q$ ” 的赋值。其中 p 是被赋值的指针变量， q 可以是指向任何变量的指针，包括栈变量、全局变量或动态分配的堆对象等。接着，我们处理指针被初始化的情况，在图7程序的第 5 行和第 11 行，使用函数 P0pmd_tbl_init_as() 插桩保存指针元数据，2 个参数分别为指针的地址和指针的值。然后，我们将原始指针的元数据传播到他们所有的派生指针。对于包括指针运算在内的指针传播，我们读取原始指针的元数据，再将其复制到派生指针的元数据中。如图7程序第 8 行，我们插入函数 P0pmd_tbl_init_or_update_as() 将指针 p 的元数据传播到指针 q 。

指针在函数间的传播涉及 2 种：1) 指针作为参数传入函数，并在函数内部传播；2) 指针作为返回值，被调用者获取，在调用者函数中传播。

UAFSan 引入投影栈支撑函数间的指针传播：如果实参包含指针，调用者在调用前将指针的元数据存入投影栈；如果返回值包含指针，被调用者将指针元数据存入投影栈。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int *p = (int*)wrapped_malloc(40); // 函数替换
5     POpmd_tbl_init_as(&p, p); // 指针采集
6     printf("addr 1: %p\n", p);
7     int *q = p;
8     POpmd_tbl_init_update_as(&q, &p); // 指针采集
9     wrapped_free(p); // 函数替换
10    int *r = (int*)wrapped_malloc(40); // 函数替换
11    POpmd_tbl_init_as(&r, r); // 指针采集
12    printf("addr 2: %p\n", r);
13    POcheck_mem_access(q, POpmd_tbl_lookup(&q)); // 内存检查
14    *q = 1;
15    wrapped_free(q); // 函数替换
16    POcheck_mem_access(r, POpmd_tbl_lookup(&r)); // 内存检查
17    *r = 1;
18 }

```

图 7. 程序编译插桩后的效果示意

3.3.3 内存检查

我们首先找到所有指针访问内存的指令，然后插入插桩函数 `POcheck_mem_access()` 以在程序执行期间检测所有可能的 UAF 漏洞。

如图7所示，在第 13 和 16 行插入函数 `POcheck_mem_access()`，因为指针 `p` 和 `r` 分别在第 14 行和第 17 行被访问。

UAFSan 的目标是堆对象上的 UAF 漏洞，因此无需检查指向非堆对象的指针。UAFSan 使用过程内的静态反向数据流分析以识别指向非堆对象的指针，并避免对此类指针检查。具体来说，给定 1 个指针指向的内存对象，找到创建该对象的指令，从中判断该对象是否是堆对象。由于该分析是保守的，我们仍可能在指针访问非堆对象前插桩检查。

3.3.4 堆对象管理器

插桩程序运行时，每个堆对象获得唯一标识符。非堆对象被单独标记：空指针或源代码未知的第三方函数返回的指针对应的对象标识符为 0，指向全局变量、静态变量、常量的指针对应的对象标识符为 1。指向栈变量的指针对应的对象标识符为 2。

UAFSan 的反向数据流分析是保守的，因此可能会对非堆对象进行检查。因此将 3 类非堆对象单独赋值，如果内存检查时发现检查的指针元数据对应的对象标识符是 0、1 或 2，就直接结束。

对于堆对象，如图8所示，我们使用全局变量 `unique_id` 作为计数器，实现堆对象标识符发生器 `get_unique_id()`。由于 0、1、2 是非堆对象的标识符，因此堆对象标识符从 3 开始自增。对象标识符是 8 字节无符号整数，一共可以标识 $(2^{64} - 3)$ 个堆对象，高于插桩程序 1 次

运行的产生的堆对象总量。

```
1 size_t unique_id = 2;      // 堆对象标识符从3开始
2 size_t get_unique_id() { // 用于产生新的堆对象标识符
3     return ++unique_id;
4 }
```

图 8. 堆对象标识符发生器

堆内存管理器由 3 部分组成：堆分配管理、释放管理和重分配管理，对应 3 类堆内存操作行为。堆分配管理实现了堆分配函数或运算符的包裹函数。C/C++ 提供了 malloc()、calloc()、mmap() 等函数以及 new()、new[]() 操作符，他们在编译插桩阶段均被替换为包裹函数。我们以 malloc() 的包裹函数为例，如图9a所示，堆分配管理操作如下：

- 1) 调用原生的 malloc() 函数。
- 2) 调用对象标识符发生器 get_unique_id()，获取 1 个新的对象标识符 obj_id。
- 3) 调用 POOmd_tbl_init_or_update_as()，将分配的对象的标识符、首地址和对象大小作为对象元数据，保存到对象元数据表中。
- 4) 调用 Porcs_tbl_update()，保存调用栈。
- 5) 返回分配的堆对象的首地址。

堆释放管理与此类似。C/C++ 释放堆内存的函数或操作符包括 free()、munmap()、delete() 和 delete[]()。我们以 free() 的包裹函数为例，如图9b所示，堆释放管理进行如下操作：

- 1) 调用内存访问检查器判断是否有 DF 漏洞。
- 2) 使用对象地址获得对象元数据。
- 3) 将对象元数据的地址置为 NULL。
- 4) 调用 Porcs_tbl_update() 记录调用栈。
- 5) 调用原生的 free() 函数。除堆内存的分配和释放外，C 语言还提供了堆重分配函数 realloc()，UAFSan 提供了相应的堆重分配管理。根据 Linux man 文档，realloc() 的原型是

```
void *realloc(void *ptr, size_t size);
```

realloc() 的作用是将 ptr 指向的堆对象的大小改为 size 字节并返回首地址，有 4 种可能的情况出现：

- 1) 如果 ptr 为 NULL，等价于 malloc(size)。
- 2) 如果 ptr 不为 NULL 且 size 为 0，等价于 free(ptr)。
- 3) 如果 ptr 不为 NULL 且 size 不为 0，glibc 不需要将堆对象移动，realloc() 不改变堆对象的首地址，仅改变堆对象的大小。
- 4) 如果 ptr 不为 NULL 且 size 不为 0，glibc 将堆对象移动：realloc() 首先分配 1 个大小为 size 的堆对象，然后将旧对象的内容拷贝到新对象，最后将旧对象释放。

针对 1) 和 2)，UAFSan 的处理与堆分配管理和释放管理相同；针对 3)，UAFSan 仅更新对象元数据中堆对象的大小；针对 4)，UAFSan 只需要先进行堆释放管理，再进行堆分配管理。但是情况 3) 需要引起重视，我们在实验中发现在特殊的漏洞程序样例下 UAFSan 漏报 UAF 漏洞。

```

1 void * wrapped_malloc(size){
2     void *ptr = malloc(size); // 调用原生malloc()函数
3     size_t obj_id = get_unique_id(); // 生成1个新的对象标识符
4     POomd_tbl_init_or_update_as(ptr, obj_id, ptr, size);
5     Porcs_tbl_update(obj_id, alocated);
6     return ptr; // 返回堆对象的首地址
7 }

```

(a) 堆分配管理算法的实现

```

1 void wrapped_free(ptr){
2     POcheck_mem_access(ptr, Popmd_tbl_lookup(&ptr)); // 内存检查
3     POmetadata *obj_metadata = POomd_tbl_lookup(ptr);
4     POomd_tbl_init_or_update_as(ptr, obj_metadata->obj_id,
5                                   NULL, obj_metadata->size);
6     porcs_tbl_update(obj_metadata -> obj_id, freed); // 调用栈
7     free(ptr);
8 }

```

(b) 堆释放管理算法的实现

图 9. 堆对象管理器

3.3.5 指针追踪器

在运行时，指针追踪器通过插桩代码管理指针元数据。具体来说，如果创建了 1 个原始指针，使用对象元数据初始化指针元数据。如果创建了派生指针，派生指针都直接或间接地继承了原始指针的元数据。特别地，如果指针传播过程中，右操作数并非指向堆对象的首地址，指针追踪器只记录首地址。例如 $p = \text{malloc}(8)$, $q = &p[2]$, 指针 q 指向了 $p[2]$, 但是指针 q 的元数据中只保存堆对象的首地址。因此，在程序执行过程中，原始指针及其所有派生指针的元数据记录了相同的值，为内存访问检查器提供了数据支撑。

3.3.6 内存访问检查器

当堆对象被访问或释放前，UAFSan 调用插桩函数 `POcheck_mem_access()` 检查是否有 UAF 漏洞。如图10，首先根据传入的指针，拿到指针元数据；又根据指针元数据中保存的堆对象地址，找到堆对象元数据。其间我们忽略传入空指针和非堆对象的情况，可能出现的情况如下 3 种：

- 1) 堆对象被释放，且再分配未发生。此时堆对象元数据中，对象地址为 `NULL`，所以堆对象非法。UAFSan 报告漏洞。
- 2) 堆对象被释放，内存再分配发生。此时堆对象元数据中的对象地址不为 `NULL`，但是指针元数据中的标识符与对象元数据的标识符不同。UAFSan 报告漏洞。
- 3) 堆对象没有被释放，其对象元数据中的对象地址不为 `NULL`，指针元数据中的对象标识符与对象元数据的标识符相同。UAFSan 不报告漏洞。

因此不论是否发生内存再分配，UAFSan 都可以发现包括 DF 在内的堆内存 UAF 漏洞。

```

1 void POcheck_mem_access(ptr, POomd_tbl_lookup(&ptr)){
2     if (ptr == NULL) return; // 忽略指针为空的情况
3     POmetadata* ptr_metadata = POpmid_tbl_lookup(&ptr);
4     if (ptr_metadata->obj_id <= 2) return; // 忽略非堆对象
5     POmetadata *obj_metadata = POomd_tbl_lookup(ptr_metadata->obj_addr);
6     if(obj_metadata->obj_address==NULL || // 死对象
7         obj_metadata->obj_id != ptr_metadata->obj_id) // 再分配
8         reportUAFError(ptr_metadata->obj_id);
9 }

```

图 10. 内存访问检查算法的实现

4 复现细节

4.1 与已有开源代码对比

我们利用桂滨法等人在论文中提供的开源代码复现 UAFSan 的工作，主要贡献如下：

1) 我们发现并修补了 UAFSan 的 2 处 bug，并提交 2 项 issue 提醒原作者修正²。第一，UAFSan 的运行库代码中有指针非法访问，在程序运行时将引发段错误导致程序崩溃；第二，论文中的内存访问检查算法没有被正确实现，使 UAFSan 无法报告悬空指针访问死对象的 UAF 漏洞如果不修正这 2 处 bug，复现实验将无法进行。猜测是作者上传代码时版本管理不当所致。

2) 恢复调用栈信息。调用栈记录了函数的每一级返回地址，动态漏洞检测工具发现漏洞后打印调用栈能给代码的维护人员提供精确的调试细节，ASan [18]、valgrind [16] 等报告漏洞时均提供调用栈信息。UAFSan 的开源代码中将调用栈记录的代码注释掉，我们的实验中都恢复他们，原因如下：第一，调用栈的记录会引起性能开销。如果不记录调用栈，与同行工作的性能对比实验将有失公平。第二，如果没有调用栈信息，我们无法确认 UAFSan 报告的 UAF 漏洞是否真是存在，不利于准确性实验。

3) 我们发现 UAFSan 存在漏报使用标准 C 库函数 realloc() 的 C/C++ 程序的 UAF 漏洞的情况，并给出了 2 种改进方案。漏报样例和改进方案细节将会在 4.4 小节详细介绍。

4.2 实验环境搭建

我们的软硬件环境如表1所示，UAFSan 是基于 LLVM 7.1.0 编译器架构的开源工具，搭建实验环境分为 2 步：

1) 搭建编译插桩模块。我们首先拉取 LLVM 的源代码，接着如图11所示，将 UAFSan 插桩模块的 lib、include 和 clang 目录替换到 LLVM 文件树的相应路径上。然后，在 LLVM 的根目录利用 cmake 配置 Makefile，参数选择如表2所示。最后，基于生成的 Makefile 编译安装。由于代码量巨大，因此编译时使用 make -j8 命令，指定 8 线程并行编译。由于在服务器上编译时间很长，1 次编译需要 1 小时以上，为防止网络断连导致编译中断，我们使用 screen 保持会话。

²<https://github.com/wsong-nj/UAFSan/issues>

表 1. 复现实验的软硬件环境

| 环境 | 参数 |
|-------|--|
| 操作系统 | Ubuntu 18.04 |
| 处理器信息 | Intel(R) Core(TM) i7-9700 CPU@3.00GHz, 3000 Mhz, 8 个内核, 8 个逻辑处理器 |
| 物理内存 | 16GB |
| 编译环境 | cmake 3.21.4, gcc 7.5.0 |

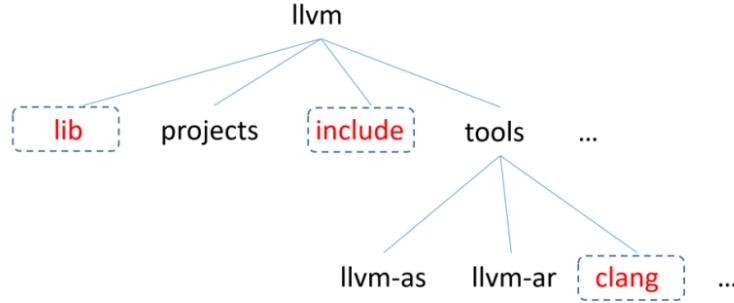


图 11. LLVM 文件树上需要被 UAFSan 的编译插桩模块替换的部分

表 2. 构建 UAFSan 的编译插桩模块的 cmake 选项

| 字段 | 值 |
|--------------------------|---------|
| -DCMAKE_BUILD_TYPE | Release |
| -DLLVM_BUILD_TESTS | OFF |
| -DLLVM_INCLUDE_TESTS | OFF |
| -DLLVM_BUILD_EXAMPLES | OFF |
| -DLLVM_INCLUDE_EXAMPLES | OFF |
| -DLLVM_ENABLE_ASSERTIONS | OFF |

2) 创建运行库。UAFSan 的运行库使用 C 语言创建。UAFSan 的 LLVM/Clang 编译器负责替换包裹函数、插入插桩函数，运行库负责为这些新引入的函数提供具体实现。配置运行库较为容易，只需要使用 cmake 的默认选项生成 Makefile，然后完成编译，即可得到静态链接库文件 libbaps.a。

4.3 界面分析与使用说明

UAFSan 被基于 LLVM 开发，使用 clang 编译器前端的编译插桩选项并链接运行库 libbaps.a，即可生成插桩程序。运行插桩程序，即可在运行时检查是否有漏洞。如图12所示，图1程序运行时发生内存重用，共有 3 处漏洞均被找出。

```
wuhui@wuhui-virtual-machine:~/UAFSan/UAFSan/runtime_library/cbaps/build$ clang -lm -fbaps ./libbaps.a -ldl -g -O0 main.c && ./a.out
addr 1: 0xa9f260
addr 2: 0xa9f260
there is a UAF occurs && it is found after memory reuse
malloc information:
#0 0x404fe7 in baps_store_malloc_back_trace_handler (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:286)
#1 0x407118 in baps_store_backtrace_metadata (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:909)
#2 0x407007 in baps_store_malloc_back_trace (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:880)
#3 0x407887 in __baps_malloc (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:1102)
#4 0x40ab1f in baps_pseudo_main (./a.out)
#5 0x404493 in main (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:132)
#6 0x7f222db09c87 in ?? (/lib/x86_64-linux-gnu/libc.so.6)
#7 0x4042aa in _start (./a.out)
free information:
#0 0x404fe7 in baps_store_free_back_trace_handler (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:299)
#1 0x40713d in baps_store_backtrace_metadata (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:913)
#2 0x40719a in baps_store_free_back_trace (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:884)
#3 0x407954 in __baps_free (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:1121)
#4 0x40ac57 in baps_pseudo_main (./a.out)
#5 0x404493 in main (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:132)
#6 0x7f222db09c87 in ?? (/lib/x86_64-linux-gnu/libc.so.6)
#7 0x4042aa in _start (./a.out)
use information:
#0 0x4050a7 in baps_store_use_back_trace_handler (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:313)
#1 0x407162 in baps_store_backtrace_metadata (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:917)
#2 0x4071ba in baps_store_use_back_trace (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:888)
#3 0x407795 in baps_pointer_dereference_check (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:1074)
#4 0x40ad63 in baps_pseudo_main (./a.out)
#5 0x404493 in main (/home/wuhui/UAFSan/UAFSan/runtime_library/cbaps/lib/baps.c:132)
#6 0x7f222db09c87 in ?? (/lib/x86_64-linux-gnu/libc.so.6)
#7 0x4042aa in _start (./a.out)
there is a UAF occurs[__baps_free] && it is found after memory reuse
malloc information:
```

图 12. UAFSan 测试图1程序漏洞的使用截图（局部）

4.4 创新点

UAFSan 对 realloc() 的处理可能导致漏报漏洞。我们首先编写带有 UAF 漏洞的样例，在实验中触发漏报的问题。然后设计 2 种针对性的改进方案：1) 改进堆重分配管理算法；2) 改进内存访问检查算法。这 2 种改进方案选其一即可解决该问题。在5.1小节中我们通过实验证明改进方案对 UAFSan 的性能几乎没有影响。

4.4.1 UAFSan 的漏报程序样例

我们设计的程序样例如图13所示。程序的执行过程如图14所示：首先，第 4 行为指针 p 分配了 40 字节的内存块，第 7 行指针 q 指向内存块的最后 4 字节，进入状态 a)。然后，第 8 行将 p 指向的内存块改为 4 字节大小，此时指针 q 成为悬空指针，进入状态 b)。最后，悬空指针 q 向其指向的区域写入 100，进入状态 c)，产生 UAF 漏洞。

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 int main(){
4     int *p = (int*)malloc(sizeof(int)*10);
5     printf("address 1: %p\n", p);
6     *p = 100;
7     int *q = &p[9];
8     p = realloc(p, sizeof(int));
9     printf("address 2: %p\n", p);
10    *q = 100; // UAF
11 }
```

图 13. UAFSan 漏报 UAF 漏洞的样例程序

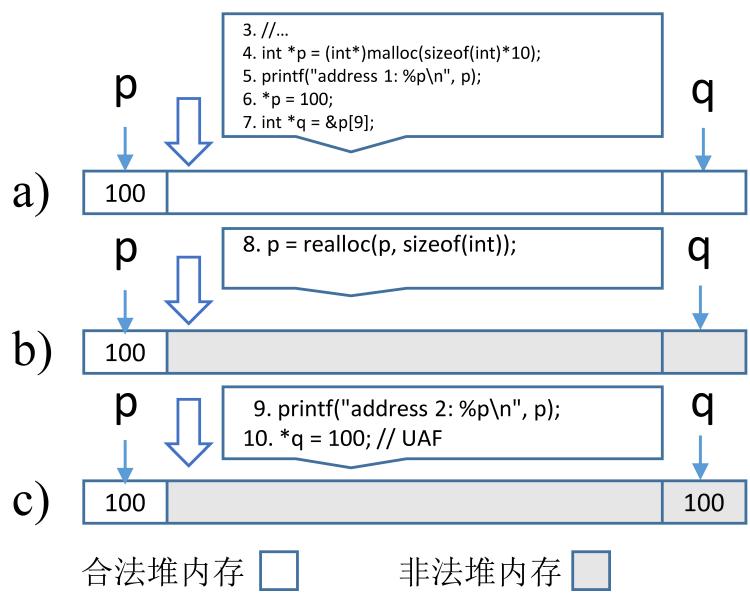


图 14. 漏报的样例程序操作堆内存的示意图

UAFSan 漏报上述 UAF 漏洞，原因如下：

- 1) UAFSan 的元数据只记录堆对象的首地址，不记录指针偏移量。调用 realloc() 后，不能判断指针是否依然合法。
- 2) UAFSan 用对象元数据的对象地址来判断是否为活对象，但是 realloc() 有时不改变对象的首地址，只改变大小。堆对象的大小改变后原本合法的指针悬空，UAFSan 对此无法有效觉察。

4.4.2 改进方案 1：改进堆重分配管理算法

针对图14所示的样例，我们重新设计堆重分配管理算法。在传入参数均不为 0 的情况下，realloc() 可能出现 2 种情况：1) 在原有的堆对象上裁剪或扩展；2) 重新分配新的堆对象，将旧对象内容拷贝到新对象，最后释放旧对象。

由于 UAFSan 不能很好地检查情况 1)，因此我们改进的堆重分配管理算法，即图15所示的包裹函数 wrapped_realloc()，强制程序进入情况 2)。

```

1 void *wrapped_realloc(ptr, size){
2     P0check_mem_access(ptr, P0pmd_tbl_lookup(&ptr));
3     P0metadata *old_obj_metadata = P0omd_tbl_lookup(ptr);
4     ret_ptr = wrapped_malloc(size); // 调用malloc的包裹函数
5     copy_size = min(size, old_obj_metadata->size);
6     memcpy(ret_ptr, ptr, copy_size); // 从旧对象拷贝到新对象
7     wrapped_free(ptr); // 释放旧对象
8     return ret_ptr;
9 }
```

图 15. 改进的堆重分配管理算法

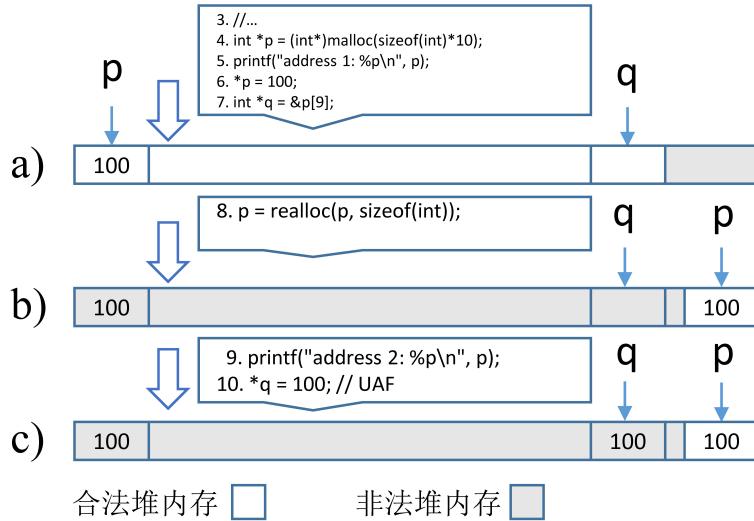


图 16. 改进方案 1 下插桩程序操作堆内存的图示

图16展示了插桩程序操作堆内存具体过程，在调用 `realloc()` 前，堆内存的进入状态 a); 然后调用 `realloc()`，包裹函数强制分配一块新的区域，并将旧区域的内容拷贝到新区域，最后释放旧区域，进入状态 b)。此时，**p** 指向一块新的内存区域的首地址，**q** 指向已经释放的内存地址。由于旧对象已经被完全释放，`wrapped_free()` 已经将旧对象的元数据的对象地址置为 `NULL`，因此再向指针 **q** 指向的内存写入 100，进入状态 c) 前，UAFSan 可以通过内存访问检查正确报告漏洞。

改进方案 1 只可能在程序调用 `realloc()` 时引入代价：glibc 裁剪或扩展对象大小只需要 $O(1)$ 的复杂度，但是拷贝旧对象的内容则需要 $O(n)$ 的复杂度。因此该改进可能对大量使用 `realloc()` 的插桩程序有性能影响。然而 C/C++ 程序中 `realloc()` 远不及 `malloc()` 和 `free()` 常用，因此在实验中我们发现改进方案 1 对插桩程序的性能影响极小。

4.4.3 改进方案 2：改进内存访问检查算法

除改进堆重分配管理算法以外，另 1 种改进方法是修改内存访问检查算法，即如图17所示，加入第 10 行的判断：在指针指向活对象的前提下，通过检查指针偏移量是否超出堆对象的大小来判断访问是否合法。

```

1 void POcheck_mem_access(ptr, POpmid_tbl_lookup(&ptr)){
2     if (ptr == NULL) return; // 忽略指针为空的情况
3     POmetadata* ptr_metadata = POpmid_tbl_lookup(&ptr);
4     if (ptr_metadata->obj_id <= 2) return; // 忽略非堆对象
5     POmetadata *obj_metadata = POomd_tbl_lookup(ptr_metadata->obj_addr);
6     if (obj_metadata->obj_address == NULL) { // ptr处是死对象
7         reportUAFError(ptr_metadata->obj_id);
8     } else if (obj_metadata->obj_id != ptr_metadata->obj_id){
9         reportUAFError(ptr_metadata->obj_id); // ptr处分配了新的对象
10    } else if (ptr - obj_address >= obj_metadata->size)
11        reportUAFError(ptr_metadata->obj_id); // 对象依然存活，但是ptr指针悬空
12 }
```

图 17. 改进的内存访问检查算法

以图14为例。状态 a) 中，对象大小为 40，指针 p 的偏移量是 0，指针 q 的偏移量是 36，p 和 q 的偏移量都没有超出对象的大小，因此可以合法访问。状态 b) 中，对象大小被改为 4，p 的偏移量是 0，小于对象大小，因此 p 可以合法访问；q 的偏移量为 36，超出对象大小，因此 q 悬空，在写入前对 q 检查，可以检测出 UAF 漏洞。

由于对所有内存访问检查都加入了对偏移量的计算和判断，该改进方案可能会带来一定的开销，但是实验证明可以忽略不计。此外，由于没有修改对重分配管理算法，因此对使用大量 realloc() 函数的程序几乎没有影响。

5 实验结果分析

我们通过实验评估 UAFSan 和改进方案的性能和准确性，主要解决以下 3 个问题：

- 1) UAFSan 检测 UAF 漏洞程序数据集和真实世界软件的 UAF 漏洞的准确性如何？
- 2) UAFSan 对基准程序数据集插桩后引入了多大的时间和内存的代价？
- 3) 我们在4.4小节中给出的改进方案是否给 UAFSan 的性能引入了额外开销？

针对问题 1)，我们使用 UAF 漏洞程序数据集 JTS 和 UAF benchmark 数据集，并复现真实世界软件漏洞测试 UAFSan 的准确性；针对问题 2) 和问题 3)，我们使用标准数据集 MiBench [9] 进行评估。

5.1 在 MiBench 上的性能评估

我们使用时间倍率 (time ratio, TR) 和内存倍率 (memory ratio, MR) 来评估 UAF 漏洞检测器运行某 1 程序的性能开销。TR 为程序插桩运行时间和不插桩运行时间的比值，MR 为程序插桩运行内存和不插桩运行内存的比值。我们使用 MiBench 中的 13 个程序评估 UAFSan 和改进方案的性能，程序中没有 UAF 漏洞。UAFSan 和改进方案的性能分析如表3所示，程序均自带大、小 2 种输入样例，测试结果相应地标有 “l” 和 “s”。

从时间代价看，valgrind 的平均 TR 为 43.04，远高于 UAFSan 的 9.45；valgrind 的时间代价更不稳定，最坏情况下 TR 高达 170.00，远高于 UAFSan 最坏情况下的 31.20。除了在 susan(l)、gsm(l) 和 dijkstra(l) 等几个程序样例下 valgrind 的时间代价更小外，其余情况 UAFSan 的时间性能均明显优于 valgrind。2 种方案均未引入明显的时间开销。由于 realloc() 在 C/C++ 程序中的使用远没有 malloc() 和 free() 频繁，realloc() 只改变堆对象大小、不改变地址的情况也只占其中一部分，因此改进堆重分配管理算法对插桩程序的 TR 影响很小。尽管 stringsearch 中用到了 realloc()，然而从 TR 看，改进堆重分配管理算法几乎没有引入时间代价。改进内存访问检查算法没有改变算法复杂度，带来的性能影响可以忽略。

从内存代价看，valgrind 的平均 MR 为 28.06，远高于 UAFSan 的 2.32。大部分情况 valgrind 的 MR 均为 30 左右，除 patricia(l)，UAFSan 的内存代价都远小于 valgrind。此外，UAFSan 的 2 种改进方案均未对 UAFSan 插桩程序的运行引入明显的内存代价。

5.2 在漏洞程序上的准确性评估

我们使用 UAF 漏洞程序数据集 JTS 和 UAF benchmark 评估 UAFSan 的准确性。如表4所示，JTS 为包含 960 个程序的 C/C++ 漏洞程序数据集。UAF benchmark 与 JTS 相

表 3. UAFSan 及其改进方案与 valgrind 的时间代价和内存代价的对比分析

| 程序样例 | UAFSan | | Valgrind | | 改进方案 1 | | 改进方案 2 | |
|-----------------|--------------|--------------|--------------|--------------|-------------|-------------|-------------|-------------|
| | TR | MR | TR | MR | TR | MR | TR | MR |
| basicmath(s) | 1.50 | 1.04 | 130.00 | 30.05 | 1.52 | 1.04 | 1.51 | 1.04 |
| basicmath(l) | 1.07 | 1.05 | 3.08 | 30.29 | 1.07 | 1.05 | 1.06 | 1.05 |
| bitcount(s) | 2.08 | 1.01 | 41.82 | 30.28 | 2.09 | 1.01 | 2.09 | 1.01 |
| bitcount(l) | 1.00 | 1.00 | 1.76 | 30.24 | 1.04 | 1.01 | 1.04 | 1.00 |
| qsort(s) | 3.67 | 1.38 | 150.00 | 23.55 | 3.67 | 1.38 | 3.68 | 1.38 |
| qsort(l) | 2.20 | 1.15 | 16.77 | 15.44 | 2.20 | 1.14 | 2.20 | 1.15 |
| susan(s) | 31.20 | 1.07 | 94.00 | 29.39 | 31.20 | 1.08 | 31.20 | 1.07 |
| susan(l) | 27.42 | 1.51 | 6.67 | 24.78 | 27.42 | 1.51 | 27.47 | 1.51 |
| adpcm(s) | 2.56 | 1.02 | 16.25 | 30.66 | 2.56 | 1.02 | 2.56 | 1.02 |
| adpcm(l) | 2.56 | 1.00 | 2.00 | 29.50 | 2.57 | 1.00 | 2.56 | 1.00 |
| CRC32(s) | 8.83 | 1.01 | 6.09 | 30.24 | 8.83 | 1.01 | 8.83 | 1.01 |
| CRC32(l) | 8.99 | 1.01 | 6.28 | 30.49 | 8.99 | 1.01 | 9.00 | 1.01 |
| FFT(s) | 4.56 | 1.03 | 51.11 | 30.30 | 4.56 | 1.03 | 4.50 | 1.03 |
| FFT(l) | 3.51 | 1.14 | 6.93 | 30.03 | 3.52 | 1.14 | 3.51 | 1.14 |
| gsm(s) | 17.67 | 1.07 | 76.67 | 30.23 | 17.67 | 1.08 | 17.69 | 1.07 |
| gsm(l) | 14.22 | 1.08 | 2.21 | 30.32 | 14.24 | 1.08 | 14.22 | 1.08 |
| sha(s) | 3.33 | 1.01 | 170.00 | 30.23 | 3.34 | 1.01 | 3.33 | 1.01 |
| sha(l) | 4.52 | 1.25 | 22.61 | 30.23 | 4.51 | 1.26 | 4.52 | 1.25 |
| blowfish(s) | 20.20 | 1.02 | 50.00 | 30.25 | 20.18 | 1.02 | 20.21 | 1.02 |
| blowfish(l) | 16.00 | 1.01 | 4.54 | 30.10 | 15.99 | 1.01 | 16.02 | 1.01 |
| stringsearch(s) | 1.00 | 1.01 | 90.00 | 30.22 | 1.00 | 1.01 | 1.00 | 1.01 |
| stringsearch(l) | 1.00 | 1.00 | 56.25 | 30.33 | 1.00 | 1.00 | 1.00 | 1.00 |
| dijkstra(s) | 23.33 | 3.18 | 50.00 | 30.47 | 23.41 | 3.18 | 23.36 | 3.19 |
| dijkstra(l) | 22.17 | 11.64 | 11.09 | 30.17 | 22.19 | 11.64 | 22.21 | 11.65 |
| patricia(s) | 11.20 | 8.98 | 45.00 | 25.06 | 11.21 | 8.99 | 11.20 | 8.98 |
| patricia(l) | 9.88 | 12.58 | 7.88 | 6.59 | 9.89 | 12.57 | 9.90 | 12.58 |
| 平均值 | 9.45 | 2.32 | 43.04 | 28.06 | 9.46 | 2.32 | 9.46 | 2.32 |

似，但其中每 1 处漏洞均发生内存再分配。UAFSan、改进方案和 valgrind 检测 JTS 和 UAF benchmark 的漏洞均未出现漏报和误报，说明 UAFSan 达到和 valgrind 同等的准确性。

表 4. JTS 和 UAF benchmark 漏洞程序数据集

| 数据集 | 类型 | 语言 | 再分配 | 无漏洞样例 | 漏洞样例 |
|---------------|-----|-----|-----|-------|------|
| JTS | UAF | C | 否 | 126 | 126 |
| | UAF | C++ | 否 | 273 | 273 |
| | DF | C | 否 | 156 | 156 |
| | DF | C++ | 否 | 404 | 404 |
| UAF benchmark | UAF | C | 是 | 126 | 126 |
| | UAF | C++ | 是 | 273 | 273 |
| | DF | C | 是 | 156 | 156 |
| | DF | C++ | 是 | 404 | 404 |

我们复现了真实世界软件的 12 个漏洞，并用 UAFSan、改进方案和 valgrind 检测。选取的软件有仅 500 行的小型教学用程序 GoHttp；也有高达 169.3 万行的大型 Linux 软件 ghostscript，用以评估 UAFSan 能否支撑真实世界不同规模的软件测试。如表 5 所示，“√”表示检测器能检测漏洞，“×”表示漏报。UAFSan 和改进方案能够检测出上述所有的漏洞，包括因内存再分配导致 valgrind 漏报的 CVE-2019-6455 漏洞。

表 5. 真实世界的软件漏洞复现测试

| 程序 | 版本 | 代码行 | 漏洞名称 | 类型 | 再分配 | UAFSan | 改进 | 改进 | valgrind |
|--------------|----------|---------|----------------------|-----|-----|--------|------|------|----------|
| | | | | | | 方案 1 | 方案 2 | 方案 1 | 方案 2 |
| nasm | 2.14.02 | 10.2 万 | CVE-2018-20535 | UAF | 是 | √ | √ | √ | √ |
| | | | CVE-2018-20538 | UAF | 否 | √ | √ | √ | √ |
| | | | CVE-2019-8343 | UAF | 否 | √ | √ | √ | √ |
| | | | CVE-2018-19216 | UAF | 否 | √ | √ | √ | √ |
| | | | CVE-2017-17820 | UAF | 是 | √ | √ | √ | √ |
| nasm | 2.14rc02 | 10.1 万 | CVE-2017-17817 | UAF | 是 | √ | √ | √ | √ |
| | | | CVE-2017-17816 | UAF | 否 | √ | √ | √ | √ |
| | | | CVE-2017-17814 | UAF | 是 | √ | √ | √ | √ |
| | | | CVE-2017-17813 | UAF | 是 | √ | √ | √ | √ |
| | | | GoHttp | DF | 否 | √ | √ | √ | √ |
| recutils | 1.8 | 4.3 万 | CVE-2019-6455 | UAF | 是 | √ | √ | √ | × |
| ghost script | 9.2 | 169.3 万 | CVE-2016-7978 | UAF | 否 | √ | √ | √ | √ |

注：UAFSan 的改进方案 1 为改进堆重分配管理算法，改进方案 2 为改进内存访问检查算法。

实验结果表明 UAFSan 具有准确和轻量级的优势；UAFSan 的 2 种改进方案检测漏洞的表现与改进前相同，但是能解决因 realloc() 引起的 UAF 漏洞漏报的问题，因此是对 UAFSan 的完善和扩展。

6 总结与展望

本文介绍了基于对象标识符的动态 UAF 漏洞检测方法，并利用桂滨法等人的开源代码在 LLVM 编译器上搭建 UAFSan，其间修补了开源代码的 2 处错误，成功复现原论文的效果。针对 realloc() 可能只裁剪堆对象的行为导致 UAFSan 漏报 UAF 漏洞，我们给出了 2 种改进方案。最后，我们通过实验在标准数据集和真实世界的软件漏洞上评估 UAFSan 及其改进方案的性能和准确性。结果证明，UAFSan 检测 UAF 漏洞具有轻量级和准确性高的优势，2 种改进方案在增加了漏洞准确性的同时，没有引入明显的性能开销。

从功能和性能的角度看，UAFSan 仍有进一步改进的空间。

首先，UAFSan 和其他基于证据的动态检测工具相比，具有轻量级和准确性高的优点，但是它的设计目前仅针对堆内存 UAF 漏洞检测。而真实世界中被广泛采用的内存漏洞检测工具如 valgrind、ASan 是以工具包的形式出现的。UAFSan 的推广可能遇到的问题之一是功能单一，而无法应对利用多种漏洞的复杂攻击模型，因此可以考虑在其框架上开发出检查其他内存漏洞的功能，如内存泄漏、越界访问、缓冲区溢出等。此外，堆内存漏洞也只是漏洞的一种，栈内存上的 UAF 等漏洞也有待支持。

其次，UAFSan 虽然能够较好地检测顺序程序的 UAF 漏洞，但是并不支撑并发场景下的 UAF 漏洞检测：对共享的数据访问没有加锁。而编写大流量、高并发的服务器端业务代码越来越是当今业界的常态，并发场景下的 UAF 漏洞需要重视。

最后，UAFSan 的性能有进一步提升的可能。插桩程序的运行时代价是由指针采集、追踪和内存访问检查的代码引起的。如果能减少指针检查的次数，则能明显提升性能。例如在一连串指令中，在没有释放堆对象的情况下，多次重复检查同一指针指向的堆对象是否合法是没有必要的，只需检查 1 次即可，循环体内也是如此。ASan [26] 采用这个思路，首先通过拆解性能实验证明内存访问检查是 ASan [18] 性能开销的主要来源，然后结合编译时的静态分析技术，减少内存访问检查的频率，大幅度降低了 ASan 的开销。这项技术同样可以应用于 UAFSan 中。

参考文献

- [1] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223. IEEE, 2011.
- [2] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. Detecting memory errors at runtime with source-level instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 341–351, 2019.
- [3] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical {Page-Permissions-Based} scheme for thwarting dangling pointers. In *26th USENIX security symposium (USENIX security 17)*, pages 815–832, 2017.

- [4] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.
- [5] Josselin Feist, Laurent Mounier, and Marie-Laure Potet. Statically detecting use after free on binary code. *Journal of Computer Virology and Hacking Techniques*, 10(3):211–217, 2014.
- [6] Andrea Fioraldi, Daniele Cono D’ Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with qasan. In *2020 IEEE Secure Development (SecDev)*, pages 23–30. IEEE, 2020.
- [7] Floris Gorter, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Dangzero: Efficient use-after-free detection via direct page table access. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1307–1322, 2022.
- [8] Binfa Gui, Wei Song, and Jeff Huang. Uafsan: an object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 309–321, 2021.
- [9] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.
- [10] Jeff Huang. Ufo: predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering*, pages 609–619, 2018.
- [11] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*. Citeseer, 2015.
- [12] Tongping Liu, Charlie Curtsinger, and Emery D Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 911–922, 2016.
- [13] Faming Lu, Mengfan Tang, Yunxia Bao, and Xiaoyu Wang. A survey of detection methods for software use-after-free vulnerability. In *International Conference of Pioneering Computer Scientists, Engineers and Educators*, pages 272–297. Springer, 2022.
- [14] Ruijie Meng, Biyun Zhu, Hao Yun, Haicheng Li, Yan Cai, and Zijiang Yang. Convul: an effective tool for detecting concurrency vulnerabilities. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1154–1157. IEEE, 2019.

- [15] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 international symposium on Memory management*, pages 31–40, 2010.
- [16] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [17] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. *ACM SIGPLAN Notices*, 53(2):377–391, 2018.
- [18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.
- [19] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.
- [20] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler: Compile-time instrumentation for threadsanitizer. In *International Conference on Runtime Verification*, pages 110–114. Springer, 2011.
- [21] Rasool Sharifi and Ashish Venkat. Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 762–775. IEEE, 2020.
- [22] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.
- [23] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.
- [24] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 42–54, 2017.
- [25] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 327–337, 2018.

- [26] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4345–4363, 2022.