Debloating Address Sanitizer

摘要

AddressSanitizer（ASan）是一款优秀的内存漏洞检测工具，能够有效地检测缓冲区溢出、释放后使用等内存漏洞，被广泛用于软件安全性测试中。然而，ASan 会带来较大的运行时开销。为了减少这一开销，ASan--结合静态分析技术，减少了 ASan 中的冗余检测，从而显著降低了 ASan 的运行时开销。

本文介绍了 ASan 的设计与实现,以及 ASan--去除冗余检测的方法,并复现了 ASan--在原论文中的实验效果。本文修正了 ASan--无法选择特定指令集架构编译的错误，并完善了漏洞数据集的测试脚本。

关键词：缓冲区溢出；释放后使用；冗余检测；动态分析；静态分析

1 引言

C/C++ 程序中，指针访问非法内存地址容易产生大量内存漏洞，比如指针越界访问会产生缓冲区溢出（buffer overflow）漏洞、指针访问已释放内存会产生释放后使用（use after free，UAF）漏洞，并且当出现以上指针非法访问时，程序并不会报出异常，而是带着已产生的漏洞继续执行。不及时检测出内存漏洞会导致程序的不稳定，出现不符合预期的结果，甚至被黑客利用，在指针非法访问的区域植入攻击程序，导致不可挽回的损失。

根据相关调研统计，截止 2019 年，在被利用来编写攻击程序的漏洞中，缓冲区溢出类漏洞最多，占比 55% [14]。根据通用漏洞披露（common vulnerabilities and exposures，CVE）数据库，UAF 漏洞数量快速增长，从 2010 年到 2022 年一直保持在较高水平 [5]。由于内存漏洞导致的数据泄露和系统崩溃事件频发，给全球数百万用户带来了巨大的数据和财产损失，因此检测内存漏洞对程序的安全稳定运行以及保障用户合法权益有着至关重要的作用。

为了精准高效地检测程序中的内存漏洞，过去的研究中提出了很多种内存漏洞检测工具，他们检测漏洞所采用的方法、检测漏洞的准确性，以及检测所带来的额外内存开销和运行时开销都有所不同。检测方法根据检测发生的时间分为动态漏洞检测和静态漏洞分析，在动态漏洞检测中，往往会产生较大的运行时开销，过去的研究中有尝试通过消除冗余检测来减少工具的运行时开销。

本文第 2 章介绍内存漏洞检测领域的相关工作，第 3 章介绍 ASan 和 ASan--的实现，第 4 章介绍 ASan--的复现细节，第 5 章为实验评估，第 6 章为总结和展望。

2 相关工作

2.1 动态漏洞检测

动态漏洞检测在程序运行时进行分析和检测，通过监视程序的执行来检测和识别可能存在的安全漏洞或异常行为，相比静态分析，动态漏洞检测在程序实际运行时捕获和分析数据，因此能够提供更真实、更全面的漏洞信息。动态漏洞检测通常借助代码插桩来进行检测，即往程序中插入相关漏洞的检测代码，在程序运行时执行该检测代码，从而收集程序信息并检测漏洞。主流的内存漏洞检测工具中，常使用编译时插桩或动态二进制插桩两种插桩方式 [3,8]。

编译时插桩是在程序运行前的编译和链接阶段，通过编译器插入检测代码，经典的编译时插桩分为两个部分：插桩模块和运行时库，运行时库为插桩模块提供内存信息。插桩模块分析程序代码以获得插桩位置，在编译时插入检测代码，在链接时运行时库会替换 malloc、free 和其他一些标准库函数，得到插桩程序，最后执行插桩程序将会在运行时进行漏洞检测，`ASan` [7] 是使用编译时插桩的经典例子，其基于编译器框架 LLVM 实现中间代码层级的插桩，并通过重写标准库函数来提供运行时库。

动态二进制插桩是在程序运行时，拦截即将执行的每条二进制指令，在拦截的指令处插入额外的检测代码，这些代码可以用于收集信息、修改程序状态或实现其他功能，插桩完成后继续执行原指令和检测代码。Valgrind [6] 和 Dr.Memory [1] 则使用动态二进制插桩，动态二进制插桩不需要源代码也能实现漏洞检测，但因为需要频繁地翻译二进制代码，会引入远大于编译时插桩的运行时开销，还可能因为二进制产生平台兼容性问题。

2.2 静态漏洞检测

静态漏洞分析在源代码或二进制代码层面进行的漏洞检测，其不需要运行程序，而是通过对静态代码进行语法分析、控制流分析、数据流分析等分析方法，检测代码中可能存在的安全漏洞或者错误。

静态漏洞分析缺少程序实际运行时数据，分析结果的误报率较高，可能会误将正常的代码结构或模式误认为是漏洞。所能检测的漏洞范围也有限，有些依赖于程序运行时上下文或者外部输入的漏洞，比如依赖于动态数据或者环境变量的漏洞，静态漏洞分析很难检测，出现大量的漏报。还难以处理复杂代码和大规模系统，比如 linux 内核和 web 浏览器，静态漏洞分析难以分析其中每个模块之间的关系和执行顺序。Klee [2] 使用符号执行来检测 UAF，并使用搜索试探法来减少路径探索，但还是会出现路径爆炸问题，只适用于较小的程序。

静态分析自身存在限制，但可以与机器学习相结合 [13,15]，比如使用机器学习提取静态分析结果中的漏洞特征来检测内存泄漏。静态分析还可以和动态漏洞检测相结合，通过静态分析代码结构提取代码的特征，再根据该特征优化插桩逻辑，`ASan--` [12] 则是通过静态分析优化了 `ASan` 插桩检测，减少了检测的次数。

动态漏洞检测通过对程序代码插桩，在程序运行时执行检测代码，而检测会带来较大的运行时开销，为了减少这一开销，一个可行的改进措施就是减少检测的次数，过去的研究中通过消除频繁执行的检测或消除冗余检测来减少动态漏洞检测的运行时开销 [9]。

2.3 消除冗余检测

ASAP [10] 去除频繁运行的代码段中的检测，减少了运行时开销但不关心去除检测后的安全性，PartiSan [4] 和 ASAP 一样为了提高性能而忽略了安全性。SANRAZOR [11] 通过捕获动态代码覆盖率和静态数据依赖关系，并使用提取的信息来分析和去除冗余检测，无法保证去除的一定是冗余检测，同样无法确保去除后的安全性。

ASan--使用纯静态的方式去除 ASan 中的冗余检测，经过理论分析可以确保去除的一定是冗余检测，而且不同于其他改进措施，ASan--在 ASan 的基础上进行改进，不改变 ASan 的使用方式，从而在减少运行时开销的同时保留了 ASan 的安全性和可用性。

3 本文方法

3.1 ASan 的设计与实现

ASan--基于 ASan 实现，ASan 由插桩模块和运行时库两个部分组成，运行时库会管理影子内存、红区、隔离队列等数据结构，插桩模块中则根据这些数据结构中的信息判断是否存在漏洞。

ASan 的工作流程如图 1 所示，ASan 插桩模块通过编写 LLVM/Clang 的 Pass 实现，LLVM 编译器前端 Clang 将程序源代码编译出中间代码 IR，然后使用 ASan 编写的 Pass 对 IR 进行处理，将检测代码插入到 IR 中，再经过 LLVM 编译器后端对 IR 继续进行编译并链接 ASan 运行时库，ASan 运行时库主要为 ASan 重写的 malloc、free 等库函数，最终生成可执行的二进制插桩程序。运行插桩程序 ASan 则会在运行时检测漏洞。

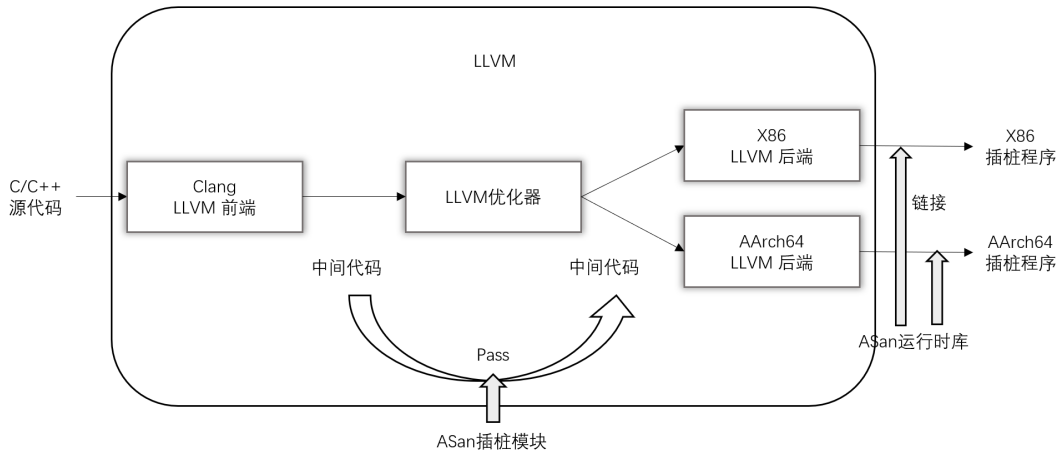

图 1. ASan 的工作流程

ASan 使用影子内存 (ShadowMemory) 记录应用程序内存的可寻址状态，其结构如图 2 所示，其使用虚拟地址空间的八分之一作为影子内存，每个字节用于记录对应八个应用程序字节的状态。在 ASan 中，对于给定的应用程序内存地址 Addr，其对应影子字节 $ShadowAddr = (Addr >> 3) + Offset$，其中 Offset 是一个由指令集架构确定的常数。

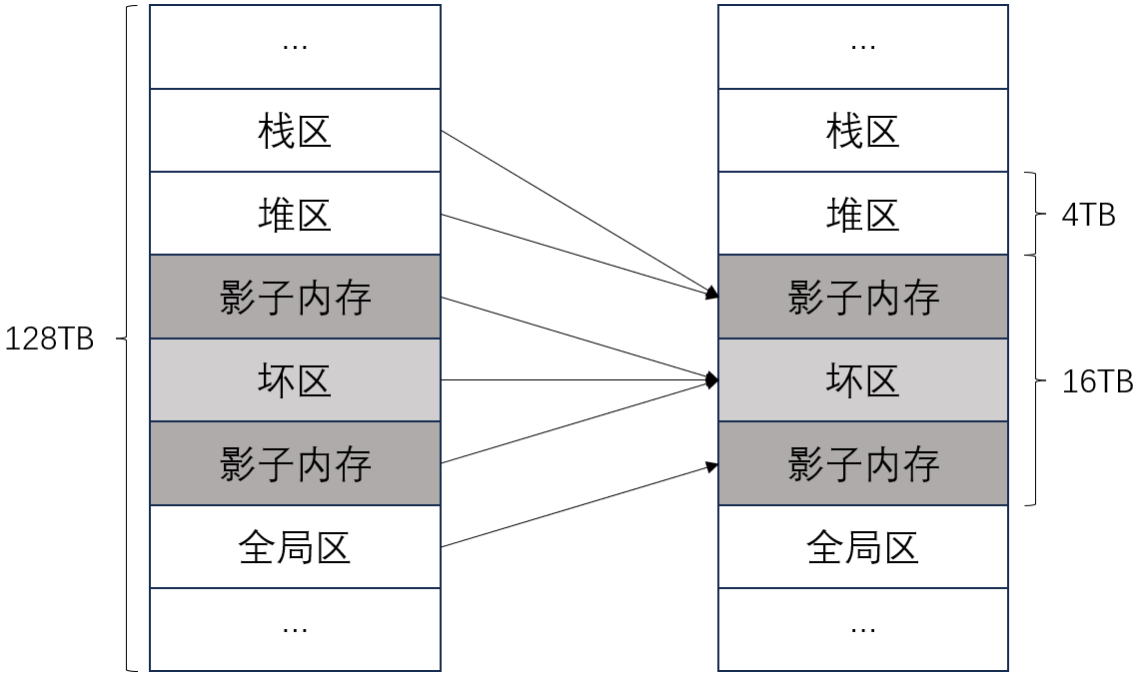

图 2. ASan 的影子内存

影子字节的值 K 用于记录相应八个应用字节的可寻址状态。当 K 为 0 时，表示这八个字节都是可寻址的；而当 K 在 1 到 7 之间时，表示只有前 K 个字节是可寻址的。负值则表示这八个字节都是不可寻址的，不同的负值则表示不同类型的不可寻址内存。

ASan 在每个对象前后各设置一个红区（Redzone），如图 3 所示，红区表示这个区域不可被寻址，对应影子字节的值也为非 0，表示完全不可寻址或者部分可寻址，越界访问对象访问到红区时即可通过影子字节的值来检测出缓冲区溢出漏洞。

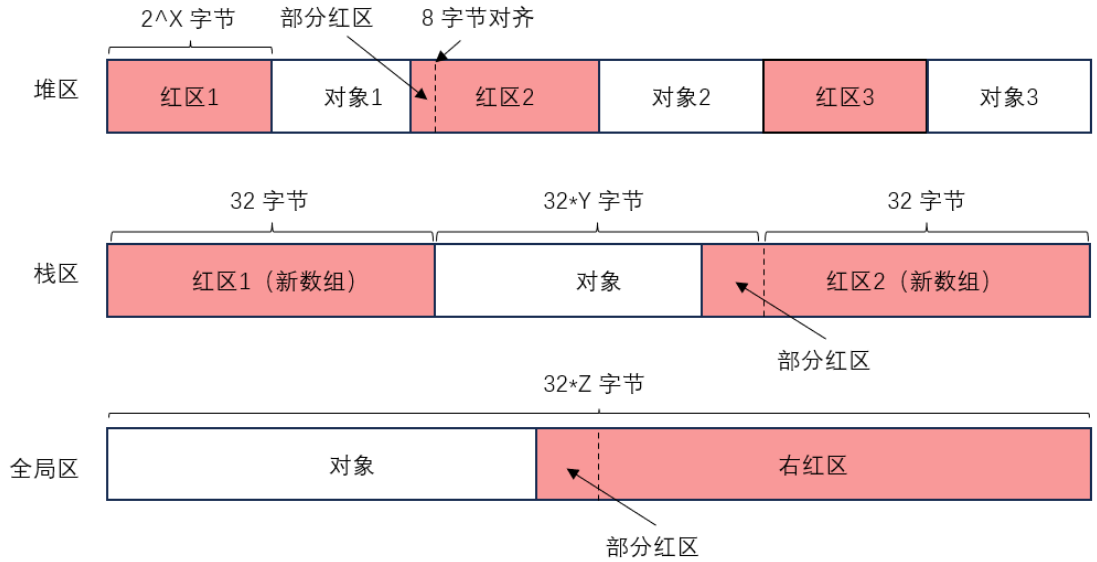

图 3. ASan 的红区。X、Y、Z 均为正整数

当使用 free 函数释放堆对象内存空间，ASan 就会将堆区相应的内存设置成红区，修改影子字节的值表示已释放内存的不可寻址，用于检测释放后使用类型的漏洞，被释放的内存空间会放入一个固定大小的隔离队列（Quarantine），当隔离队列满了后会让最先进入队列的内

存空间出队，并去除与这块内存空间相关的红区，以重新让应用程序使用。

ASan 的插桩检测是基于 LLVM/Clang 在编译时识别程序访问内存的代码，并在该代码前插入对影子内存的检查，比如在指针读写前插入检查，根据检查的结果判断是否已检测出漏洞。按照读写的大小，检查的工作方式有所不同。

对于 8 字节的读写，ASan 检查其影子字节是否为零，如代码 1 所示：

Listing 1: ASan 检查 8 字节读写

```
ShadowAddr = (Addr >> 3) + Offset;
if (*ShadowAddr != 0)
    ReportAndCrash(Addr);
```

对于 N 字节（N = 1、2 或 4）的读写，ASan 检查其影子字节是否表示前 N 字节可寻址，如代码 2 所示：

Listing 2: ASan 检查 1、2、4 字节读写

```
ShadowAddr = (Addr >> 3) + Offset;
V = *ShadowAddr;
if (V != 0 && ((Addr & 7) + N) > V)
    ReportAndCrash(Addr);
```

3.2 ASan--消除冗余检测

不同于 ASan 每次内存访问都进行插桩检测，ASan--基于 LLVM 对代码结构进行静态分析，去除了一些不必要的插桩检测，或者将多次插桩检测合并成一次。ASan--从四个方向消除冗余检测，分别是去除不可满足检查、去除重复检查、优化相邻检查、优化循环检查。

3.2.1 去除不可满足检查

不可满足检查是指一定检测不出漏洞的检查，如代码 3 所示（红色部分是 ASan 插入的检查，后文中同理），变量 i 初始化为常数 10，$i++$ 后为 11，数组 buf 大小为 20，$11 < 20$，在编译阶段便可以分析出 $buf[i]$ 肯定不会出现越界访问等漏洞，可以去除对 buf[i] 的检测。

Listing 3: 不可满足检查

```
char buf[20];
unsigned int i = 10;
i++;
ASan(buf + i);
buf[i] = 0;
```

还有一种扩展的情况如代码 4 所示，变量 i 初始不是常数，但 i 与常数 10 进行了比较，可知条件语句内的 i 必定小于 10，从而小于数组大小 20，也可在编译阶段就分析出 $buf[i]$ 不存在漏洞，可以移除对 $buf[i]$ 的检查。

Listing 4: 不可满足检查扩展

```
char buf[20];
unsigned int i = input();
if (i <= 10) {
    ASan(buf + i);
    buf[i] = 0;
}
```

3.2.2 去除重复检查

重复检查是指对同一对象进行多次检查，如代码 5 所示，$*p == 0$ 访问 p，$*p = 1$ 再次访问 p，ASan 在两次的访问前都会对 p 进行检查，而在编译时便可分析出两次访问的是同一对象，得知第二次检查是多余的，可以被移除。

Listing 5: 重复检查

```
int *p;
ASan(p);
if (*p == 0) {
    ASan(p);
    *p = 1;
}
```

3.2.3 优化邻居检查

邻居检查是指对两个相邻的对象进行检查，优化邻居检查包括两种情况，分别是合并邻居检查和移除邻居检查。

合并邻居检查如代码 6 所示，结构体中 a 和 b 为邻居，代码中分别访问了 a 和 b，ASan 会对 a 和 b 分别进行检查，但因为 a 和 b 相邻，且都是整型，共占用 8 字节的连续内存空间，而应用内存和影子内存是 8 比 1 的映射关系，考虑不对齐的因素，a 和 b 对应 1 个或 2 个影子字节。

Listing 6: 可合并邻居检查

```
struct S {
    int a, b;
};
int main() {
    struct S *p;
    ASan(&p->a);
    p->a = 1;
    ASan(&p->b);
    p->b = 2;
}
```

short 类型为 2 个字节，可以覆盖 a 和 b 对应的影子字节，因此可以修改对 a 的检查如代码 7 所示，使用 short 类型指针一次读取 2 个影子字节，判断是否为 0，如果为 0 则说明 a 和 b 均不在红区中，没有漏洞；如果非 0，则说明 a 和 b 都有可能落入红区，再分别对 a 和 b 进行 ASan 中的原始检查。因为程序一般是没有漏洞的，因此判断的结果基本都为 0，从而基本都能从原来读取 2 次内存合并成只用读取 1 次内存，便能减少时间开销。

Listing 7: 合并邻居检查

```
short *ShadowAddr;
ShadowAddr = (&p->a >> 3) + Offset;
if (*ShadowAddr != 0) {
    second_check(&p->a);
    second_check(&p->b);
}
```

移除邻居检查如代码 8 所示，a、b 和 c 是邻居，代码中分别访问了 a、b 和 c，假设 b 在红区中，由于红区大小最小为 16 字节且 b 只占 4 字节，那么与 b 相邻的 a 和 c 肯定至少有一个也在红区中，因此在 b 中出现的漏洞肯定可以在 a 或 c 的检查中捕获到，则可以去除对 b 的检查。

Listing 8: 可移除邻居检查

```
struct S {
    int a, b, c;
};
int main() {
    struct S *p;
    ASan(&p->a);
    p->a = 1;
    ASan(&p->b);
    p->b = 2;
    ASan(&p->c);
    p->c = 3;
}
```

3.2.4 去除不可满足检查

循环检查是指在循环结构中的检查，优化循环检查有两种情况，分别是重定位不变检查和分组单调检查。

不变检查是指在循环中重复对同一对象进行检查，如代码 9 所示，循环中每次迭代都会对访问同一对象，ASan 每次会检查该对象，但这是存在大量冗余的，例如，如果循环中的一个数组元素在第一次迭代中被检查为安全，那么在后续迭代中再次检查这个元素就是不必要的，因此可以将循环中的检查重定位移出到循环结束后，只有在循环结束后检查一次，而不是在每次迭代中都进行检查。。

Listing 9: 不变检查

```
char *p;
for (int i = 0; i < end; i += 2) {
    ASan(p);
    *p = getchar();
}
```

单调检查是指循环中每两次检查之间相隔固定的内存空间，如代码 10 所示，p 是不变值，循环每次迭代都会使 $i+=2$，因此循环中每两次访问 $p[i]$ 之间相隔 2 个字节，虽然两次访问的内存空间不一定相邻，但可以使用前面合并相邻检查类似的思想来对单调检查进行分组检查。

Listing 10: 单调检查

```
char *p;
for (int i = 0; i < end; i += 2) {
    ASan(p + i);
    p[i] = getchar();
}
```

每次访问目标地址 Addr 前，不执行 `ASan` 原先的检查，而是执行代码 11 中的检查，Init 是单调检查的初始地址，条件语句将最小红区大小的内存空间合并到了一组，使用能覆盖最小红区对应影子内存大小的指针一次读取这一组对应影子内存的值，如果值不为 0 则说明这一组都没有落入红区中，内存访问均不会存在漏洞；如果不为 0 则使用 `ASan` 检查 Addr。这同样由于程序中基本不会有漏洞，因此基本可以从读多次内存合并成只读一次内存。

Listing 11: 分组单调检查

```
intN *ShadowAddr;
if ((Addr - Init) % MinRdSz < Step) {
    ShadowAddr = (Addr >> 3) + Offset;
    if (*ShadowAddr != 0)
        ASan(Addr);
}
```

在循环退出的时候，如果循环最后一次迭代的 Addr 和 Init 不同，还会使用 `ASan` 检查一次 Addr，因为如果 Addr 没有按 8 字节对齐，且 Addr 所在的组末尾有 1 字节的红区，而 $Addr >> 3$ 会舍去末尾的红区，读取影子内存时则无法检测出红区，导致出现漏报的情况，因此需要末尾的访问需要额外一次检查。

4 复现细节

4.1 与已有开源代码对比

4.1.1 修正编译错误

在构建 LLVM 时，为了减少磁盘占用，添加了一个 LLVM 自定义的 CMake 构建选项 *LLVM_TARGETS_TO_BUILD*，从而只构建指定指令集架构的可执行文件，可满足单一实验环境的使用，但添加该选项编译出来的 Clang 在使用 `ASan--`编译程序时会编译失败，部分报错信息如代码 12 所示。

Listing 12: ASan–存在的错误

```
Pass 'AddressSanitizerFunctionPass' is not initialized.
Verify if there is a pass dependency cycle.
Required Passes:
    Dominator Tree Construction
    Target Library Information
```

经过研究 `ASan` 的源代码以及 `ASan--`提交的修改，发现错误出现在 `ASan--`在进行去除重复检查优化时，为 `ASan` 引入了 LLVM 内置的两个依赖 Pass（PostDominatorTree 和 AAResults），这两个依赖分别可以提供代码的支配树分析和别名分析，`ASan--`根据分析的结果决定是否可以移除某些重复的检查。

如图 4 所示，`ASan` 的插桩模块也是一个 Pass，LLVM 中的 Pass 机制允许使用内置 Pass 和自定义 Pass，在编译器中间代码优化阶段分析或转换中间代码，比如 `ASan` 是自定义 Pass，可以转换中间代码，上述两个内置 Pass 则可以分析中间代码。LLVM 还允许为 Pass 设置依赖 Pass，以使用其他 Pass 的功能，比如 `ASan--`就添加了这两个有分析功能的 Pass 作为依赖。LLVM 中所有的 Pass 都需要初始化后才可使用，且会根据依赖关系来决定初始化的顺序，Pass 的生命周期有限，比如如果中间代码被修改，基于原先中间代码的分析将失效，相应的 Pass 被销毁。如果 Pass 已被销毁但仍想继续使用，可以重新初始化该 Pass。

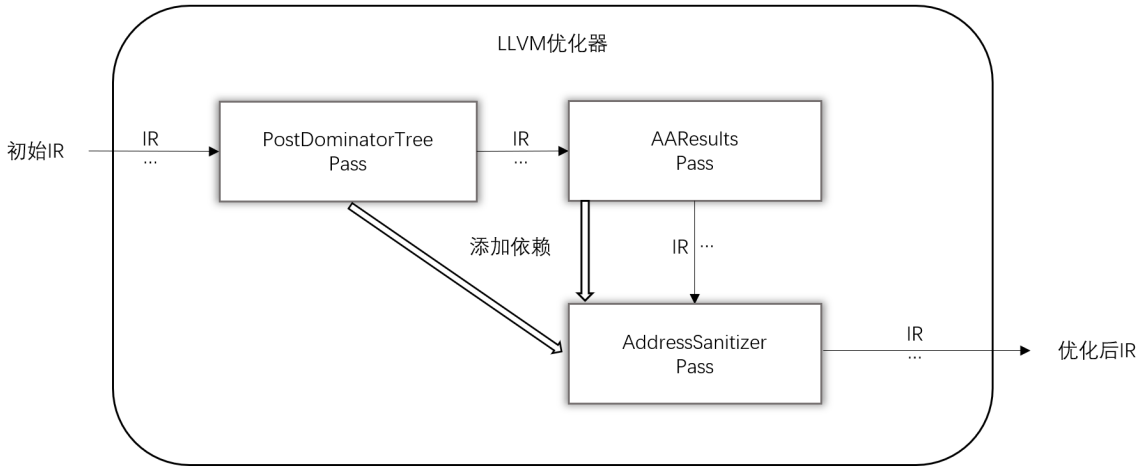

图 4. LLVM Pass 的工作流程和依赖关系

ASan--添加了依赖，如果不添加 $LLVM\_TARGETS\_TO\_BUILD$ 进行构建，编译插桩程序时依赖在 ASan--初始化前还未被销毁，因此可以正常初始化 ASan--。而添加之后依赖在 ASan--初始化前已经被销毁，所以 ASan--会因缺乏依赖而导致初始化失败。本文在 ASan--初始化前重新初始化一次新添加的依赖，确保其依赖不被销毁，修正后 ASan--可正常使用。

4.1.2 完善测试脚本

官方提供的 Juliet Test Suite（JTS）测试脚本需要人工逐个测试样例进行测试，效率较低，因此本文基于一个更高效的开源测试脚本，对某些特殊的测试样例添加了判断，提高了测试结果的稳定性。

为了保证每次运行 JTS 时都能尽可能获得相同的测试结果，需要消除随机数 $rand()$ 的影响，因此将 JTS 数据集中 $globalReturnsTrueOrFalse()$ 的返回值更改为 1，还过滤掉了以下三个测试：

1）名称中包含 *socket* 的测试需要在客户端和服务器上运行。它们不适用于 ASan--测试，并且会带来不确定的超时，从而导致测试结果不一致。

2）名称中包含 *rand* 的测试也会因为存在随机数而导致测试结果不一致。

3）名称包含 $CWE170\_char\_*$ 的测试打印不带终止字符的字符串，这将产生随机溢出。

为了实现自动化测试，对于一般输入，通过读取文件统一传入 11。对于下溢读写（CWE124、CWE127），通过读取文件统一传入 -1。

4.2 实验环境搭建

ASan--以组件的形式在 LLVM 中存在，可以选择构建，两者在 LLVM 整个项目中的组成如图 5 所示。

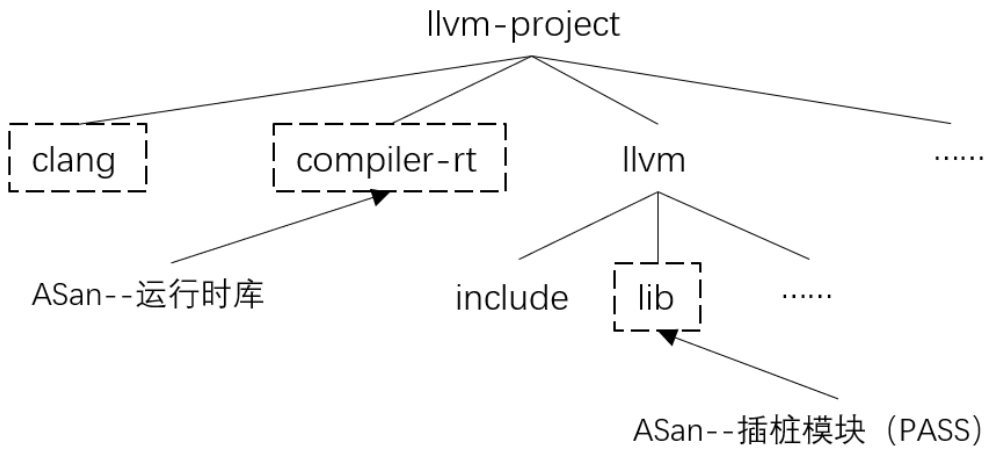

图 5. LLVM 项目结构

本文使用 ASan--的源代码，源代码为对 ASan 修改后的 LLVM 4.0 项目，通过命令行构建，步骤如下：

1）在 LLVM 根目录下使用命令 mkdir ASan--Build && cd ASan--Build 创建并进入构建目录。

2）构建 LLVM，并添加构建 Clang 和 ASan--，添加 CMake 构建选项如代码 13 所示，选择 Release 进行构建减少磁盘占用，选择只构建实验环境指令集架构所需的二进制文件进一步减少不必要的磁盘占用。

Listing 13: 构建 Clang 和 ASan–

```
$ cmake -G "Unix Makefiles" \
-DLLVM_ENABLE_PROJECTS="clang;compiler-rt"\
-DCMAKE_BUILD_TYPE=Release \
-DLLVM_TARGETS_TO_BUILD=X86 ../llvm
```

3）使用命令 $make - j4$ 多线程编译。

4.3 使用说明

使用构建好的 Clang 编译 C/C++ 代码，并添加编译选项以启用 ASan--，例如./bin/clang -fsanitize=address buggy.c -o buggy，运行插桩程序./buggy，ASan--则能检测出代码中的漏洞。

5 实验结果分析

本文实验环境为在 12 核 CPU 和 32GB 内存的 X86 平台，使用 ubuntu18.04 的 Docker 容器，通过实验评估 ASan 和 ASan--的运行时开销以及检测漏洞的准确性，并分别使用不同数据集进行评估：

1)对于运行时开销，本文使用基准测试集 SPEC CPU2006，分别测试启用 ASan 和 ASan--后 Clang 所编译基准程序的运行时间，并与不启用情况下的运行时间做对比。

2）对于检测漏洞的准确性，本文使用 Juliet Test Suite 漏洞数据集和 Linux Flaw Project 中的 CVE 漏洞，测试 ASan 和 ASan--能否检测出数据集中的漏洞。

5.1 运行时开销

本文使用 SEPC CPU2006 评估运行时开销，SEPC CPU2006 是商用的计算密集型的基准测试套件，包含两个套件，CINT2006 套件测量计算密集型整数性能，CFP2006 套件测量计算密集型浮点性能。每个套件都有三种不同规模且用于不同场景的基准数据集，从小到大分别为 test、train 和 reference，每个基准数据集包含若干个基准程序，基准程序以源代码的形式提供。

不同基准程序的源代码可能使用不同语言编写，共有三种语言 C、C++ 和 Fortran，因为需要使用 Clang 编译源代码，无法编译 Fortran 代码，因此使用 Fortran 编写的基准程序不包含在测试中。基准程序 omnetpp 会与本文使用的 LLVM/Clang 4.0.0 中的 ASan 冲突，无法通过编译，和 ASan--团队的实验一样，本文也不对其进行测试。

本文使用 CINT2006 和 CFP2006 中 reference 规模的数据集，并使用 Clang 编译数据集中每个基准程序的源代码，依次运行编译出的可执行程序。使用三种编译方式进行编译：1）只开启 O2 优化；2）开启 O2 优化并启用 ASan；3）开启 O2 优化并启用 ASan--。统计三种编译方式下各自的运行时间，各运行三次取运行时间的中位数。

由于本文使用的 SEPC CPU2006 本身可能包含漏洞，而 `ASan` 和 `ASan--`默认情况下在运行时检测到漏洞会终止程序，无法统计完整的运行时间，因此本文设置了检测到漏洞不终止，即在运行时添加环境变量 `ASan`_OPTIONS=halt_on_error=0。

接下来计算 `ASan` 和 `ASan--`的运行时开销，上面已获得三种编译方式下程序的运行时间，使用公式 1 分别计算 `ASan` 和 `ASan--`的运行时开销。

$$\text{Overhead} = \frac{\text{Time}_{02} - \text{Time}}{\text{Time}_{02}} \times 100\% \tag{1}$$

如表 1 所示，`ASan` 平均运行时开销为 110.30%，而 `ASan--`经过消除冗余检测后将平均运行时开销减到 77.72%，减少了约 33%，基准程序 hmmer 中的运行时开销更是从 173.82% 减到 97.82%，减少了 76%，优化十分明显。

表 1. SPEC CPU2006 测试数据

基准程序	`ASan`	`ASan--`
400.perlbench	256.07%	192.50%
401.bzip2	63.85%	45.07%
403.gcc	157.33%	140.44%
429.mcf	52.60%	39.14%
445.gobmk	88.41%	47.26%
456.hmmer	173.82%	97.82%
458.sjeng	79.16%	45.12%
462.libquantum	35.66%	9.09%
464.h264ref	123.55%	97.97%
473.astar	52.88%	30.96%
483.xalancbmk	113.93%	99.59%
433.milc	112.27%	54.55%
444.namd	94.78%	76.09%
447.dealII	200.53%	137.89%
450.soplex	59.07%	51.48%
453.povray	239.62%	168.87%
470.lbm	20.72%	5.86%
482.sphinx3	61.15%	59.31%
Average	110.30%	77.72%

5.2 检测漏洞准确性

本文使用开源漏洞数据集 Juliet Test Suite 评估检测人工设计漏洞的准确性，Juliet Test Suite 包含大量使用 C/C++ 编写的漏洞样例，少数样例只有 bad() 函数，大多数样例都包含一个 good() 函数和一个 bad() 函数，bad() 是存在漏洞的代码，good() 是修复 bad() 中漏洞后的代码，代码结构和 bad() 相似。

本文使用 Clang 编译运行漏洞样例程序，并分别启用 ASan 和 ASan--。程序运行时，如果 good() 函数中未检测出漏洞，说明没有出现误报，好测试 pass 的总数加 1；如果 bad() 函数中检测出漏洞，说明没有出现漏报，坏测试 pass 的总数加 1。如表 2 所示，ASan 和 ASan--在测试中得到完全相同的结果，说明 ASan--保留了 ASan 检测漏洞的准确性。

表 2. Juliet Test Suite 测试数据

CWE(编号)	好测试 (通过数/总数)	坏测试 (通过数/总数)
Stack Based Buffer Overflow (121)	2956/2956	2948/2956
Heap Based Buffer Overflow (122)	3582/3582	3390/3582
Buffer Underwrite (124)	1024/1024	1024/1024
Buffer Overread (126)	672/672	672/672
Buffer Underread (127)	1024/1024	1024/1024

本文使用开源项目 Linux Flaw Project 评估检测真实世界软件漏洞的准确性，Linux Flaw Project 收录了大量 CVE 漏洞及其复现方式，本文选择了与 ASan 和 ASan--相关的 11 个 CVE 漏洞，涵盖 6 个真实世界软件的不同版本，分别使用 ASan 和 ASan--看能否检测出漏洞，实验中两者都能检测出 11 个漏洞，如表 3 所示，进一步说明 ASan--保留了 ASan 检测漏洞的准确性。

表 3. CVE 漏洞复现情况

软件	版本	CVE 编号	类型	ASan	ASan--
fcron	3.0.0	CVE-2006-0539	heap-buffer-overflow	✓	✓
ctorrent	3.3.2	CVE-2009-1759	stack-buffer-overflow	✓	✓
libzip	1.2.0	CVE-2017-12858	use-after-free	✓	✓
		CVE-2013-4473	stack smashing	✓	✓
poppler	0.24.2	CVE-2013-4474	stack-buffer-overflow	✓	✓
	2.15	CVE-2006-2362	stack-buffer-overflow	✓	✓
binutils	2.29	CVE-2018-9138	stack-overflow	✓	✓
	3.8.0	CVE-2006-2025	Integer-overflow	✓	✓
libtiff	3.8.2	CVE-2009-2285	heap-buffer-overflow	✓	✓
	4.0.1	CVE-2013-4243	heap-buffer-overflow	✓	✓
		CVE-2015-8668	heap-buffer-overflow	✓	✓

6 总结与展望

本文介绍了内存漏洞检测的检测方法，以及消除冗余检测的优化方法，还介绍了 ASan 和 ASan--的实现。本文搭建了 ASan--，还修正了 ASan--中的一个错误。本文还进行了实验，使用 SPEC CPU2006 评估 ASan--的运行时开销，使用 Juliet Test Suite 和 Linux Flaw

Project 评估了 ASan--检测漏洞的准确性，说明 ASan--在减少 ASan 运行时开销的同时还保留了 ASan 检测漏洞的准确性。

ASan--需要支持更高的 LLVM 版本，ASan--的优化不改变 ASan 的使用方式，原先基于 ASan 实现的工具可以很方便地整合 ASan--的优化，从而直接减少工具的运行时开销，但 ASan--目前基于 LLVM 4.0.0 实现，4.0.0 已经是数年前的版本，本文编辑时 LLVM 版本已经到了 19.1.6，ASan--版本太过落后，如果想将 ASan--和其他工具相结合，可能会因 LLVM 的 API 相差太大导致冲突，因此 ASan--需要进行高版本迁移。

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] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[3] Binfa Gui, Wei Song, Hailong Xiong, and Jeff Huang. Automated use-after-free detection and exploit mitigation: How far have we gone? *IEEE Transactions on Software Engineering*, 48(11):4569–4589, 2021.

[4] Julian Lettner, Dokyung Song, Taemin Park, Per Larsen, Stijn Volckaert, and Michael Franz. Partisan: fast and flexible sanitization via run-time partitioning. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*, pages 403–422. Springer, 2018.

[5] 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.

[6] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[7] 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.

[8] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[9] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[10] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

[11] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 479–494, 2021.

[12] 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.

[13] 王林章 李宣东朱亚伟, 左志强. C 程序内存泄漏智能化检测方法. 软件学报, 30(5):1330, 2019.

[14] 苏璞睿, 黄桦烽, 余媛萍, and 张涛. 软件漏洞自动利用研究综述. 广州大学学报 *(*自然科学版*)*, 18(3):52–58, 2019.

[15] 黄建军游伟石文昌 张健边攀, 梁彬. Rtdminer: 基于数据挖掘的引用计数更新缺陷检测方法. 软件学报, 34(10):4724, 10 2023.