

Floatzone: 使用浮点运算单元加速内存漏洞检测

摘要

内存检测器 (sanitizers) 是强大的工具, 用于检测空间和时间相关的内存错误, 例如缓冲区溢出和释放后使用 (use-after-free)。然而, 检测器通常会带来显著的运行时开销。例如, 最广泛使用的检测器 Address Sanitizer (ASan) 会导致程序运行速度减慢两倍。主要的性能开销来自于检测器的检查操作, 这些操作至少包括一次内存查找、一项比较以及一个条件分支指令。将这些检查应用于程序的所有内存访问验证会大幅降低执行效率。

本文复现了 Floatzone, 一种基于编译器的检测器, 旨在通过轻量化的检查检测 C/C++ 程序中的空间和时间内存错误。Floatzone 利用浮点运算单元 (FPU), 将“查找、比较和分支”的组合效果整合为一次浮点加法操作, 该操作在发生内存违规时触发下溢异常。这种新颖的方法通过避免传统比较的缺陷显著提高了性能: 它防止了分支预测错误, 提高了指令级并行性 (通过将操作卸载到 FPU), 并减少了缓存未命中率 (因不需要影子内存)。

复现结果表明, Floatzone 显著优于现有系统, 在 SPEC CPU2006 基准测试中仅带来 31% 的运行时开销。最后, 我们确认 Floatzone 的检测能力与 ASan 相当, 测试数据包括 Juliet 测试套件和一组 CVE 漏洞。

关键词: Sanitizer; ASan; Shadow Memory; Redzone; Float Point Unit Exception

1 引言

内存安全问题一直是软件安全领域中最为核心且久未解决的挑战之一, 特别是在使用 C/C++ 等非内存安全语言编写的系统软件中, 与内存读写相关的安全漏洞频繁出现。这些漏洞不仅会导致缓冲区溢出 (Buffer Overflow) 和 Use-After-Free 等典型问题, 还可能引发系统崩溃、数据泄露, 甚至远程代码执行, 对软件的稳定性和安全性构成了严重威胁。在关键基础设施和高安全性环境中, 随着软件系统复杂性的不断提升, 内存安全的重要性越发突出。

为了应对这一挑战, 研究者们提出了多种内存错误检测和防御方法 [4,7,9,20]。其中基于编译器插桩的工具如 AddressSanitizer (ASan) [21] 因其卓越的检测能力和广泛适用性, 得到了业界的高度重视。ASan 通过在程序编译期间对其设置红区 (redzones) 和插桩 (instrument) 监控内存读写操作的指令, 能够有效检测诸如越界访问、下溢以及时间维度的错误。然而这类工具在实现高检测覆盖率的同时, 也不可避免地引入了显著的运行时开销, 对性能敏感场景 (如模糊测试) 的效率造成了严重影响。

为降低运行时开销并平衡检测覆盖范围, 研究者们探索了多种优化路径。例如通过精简检测指令 [16,23,24]、优化元数据管理 [8,10] 或借助硬件扩展 [17,22] 来降低成本, 目前已经取得了一定进展。此外还有研究尝试对局部错误进行低开销检测 [5,7], 以满足特定场景下的

性能需求。然而，对于需要兼顾时间和空间两个维度全面内存保护的场景，检测指令依然是性能瓶颈的主要来源 [19]，最近的分析显示 ASan 的开销中约有 80% 是由检查引起的 [24]。在缺乏硬件支持的环境中，检查安全属性通常依赖于比较操作与分支指令的结合。以 ASan 为例，每次加载或存储时，都会定位相关元数据，通过比较判断是否落入红区。如果检测到违规，程序会跳转到负责触发警报的退出代码。这种方式不仅使分支预测器的效率受到影响，还导致 CPU 执行单元资源被分散，同时对影子内存的访问进一步加剧了缓存和 TLB 的压力。

针对当前内存安全检测工具的痛点，本次复现的论文提出了一种无分支、隐式内存错误检测的新方法，这个方法被称为论文称之为 Floatzone。它利用现代 CPU 的浮点功能单元，通过浮点指令在非法内存访问发生时触发异常，将检测压力分担到在多数应用中通常未被高效使用的浮点执行单元上，最大化利用指令级并行度。Floatzone 还摒弃了传统的影子内存管理，采用在红区内嵌入标记值的方式来检测并区分非法访问。同时 Floatzone 在快速验证机制减少潜在误报的基础上，结合现代内存隔离技术，实现了对时间维度内存错误的有效防护。

本次复现论文中的实验结果表明，Floatzone 在多个测试场景中均表现出了显著的性能优势，这与本次复现的结果是一致的。在 SPEC CPU 基准测试和多组模糊测试中，Floatzone 的运行时开销较 ASan 显著降低，平均吞吐量提升数倍，同时保持了较高的内存安全覆盖范围。这些成果表明，随着硬件性能与指令集的持续发展，通过合理利用 CPU 功能单元分担检测任务，不仅为内存安全提供了新的优化思路，也为软件安全工具的高效实现开辟了新路径。

2 相关工作

2.1 内存检测工具

内存检测工具 (Memory sanitizers) 是一类用于检测以不安全语言 (如 C 和 C++) 开发的程序中内存违规问题的工具。它们提供的内存安全保障通常分为两大类：

1. 时间安全性 (Temporal safety)：所有对对象的内存访问必须发生在其生命周期内。例如，使用释放后的内存 (use-after-free) 和重复释放 (double-free) 问题都违反了时间安全性；
2. 空间安全性 (Spatial safety)：所有内存访问必须在引用对象的边界范围内。例如，堆和栈缓冲区溢出就是空间安全性违规的典型问题。

实现内存安全的常见方法是使用红区 (redzones)。这种方法在内存对象之间插入填充区域，检测工具确保对红区的内存访问会触发错误，从而检测那些未越过红区的空间内存错误。同样地，被释放的内存也可以用红区来保护，以便在重新分配之前检测时间内存错误。检测工具在每次内存访问时都会进行运行时的有效性检查。这里的“红区”一词是广义的，包括了已释放的内存。最为广泛应用的是 ASan 的红区管理方案，其分配如图 1 所示。对于不同类型的内存对象，ASan 采取不同的红区管理措施：

1. 对于堆区，一旦堆缓冲区被分配，ASan 会在缓冲区的前后各放置一个红区。为了提高效率，ASan 会将多个相同大小的缓冲区作为一组分配，并将它们挨在一起。一个缓冲区的左侧红区作为前一个缓冲区的右侧红区。这样，ASan 只需要为从该组分配的缓冲

区分配一个红区（左侧红区）。一旦缓冲区被释放，ASan 会将整个缓冲区标记为无效并将其放入一定大小的隔离区中，以便检测 Use-After-Free 错误；

2. 对于栈区，在堆上每个数组的前后，ASan 会插入两个相邻的数组，分别作为左侧和右侧红区；
3. 对于全局变量，ASan 会用一个新对象替换每个全局对象，该对象包含一个尾部红区，所有全局对象的红区在进程初始化时都会被标记为无效。

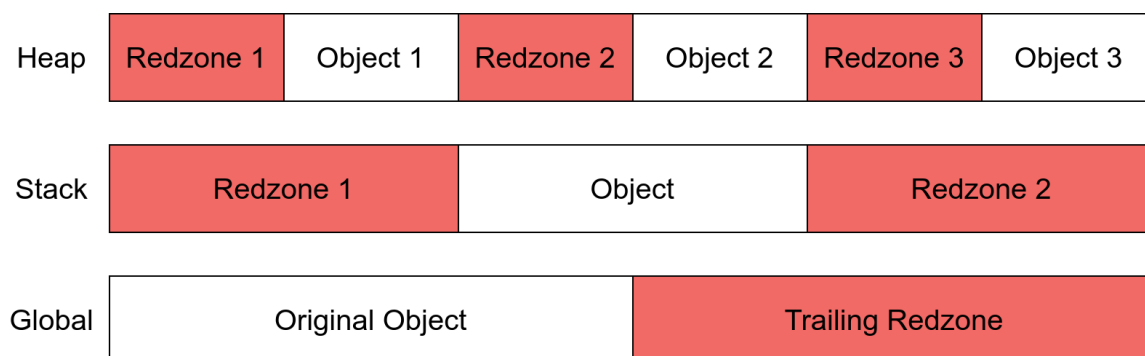


图 1. ASan 在堆区，栈区，全局区上面的红区分配

传统上，检测工具依赖于影子内存（shadow memory），这是一个独立的内存区域，用于存储应用程序中红区的元数据，提供关于哪些内存区域可访问的“真实信息”。影子内存需要在内存分配和释放时进行相应管理。

ASan 使用一种影子内存模型（如图 2 所示）来实现高效的运行时检查。它将虚拟地址空间的八分之一保留为影子内存，其中每个字节记录应用程序中八个字节的可访问状态。对于地址为 Addr 的内存字节，ASan 将其对应的影子字节放置在 $(\text{Addr} \gg 3) + \text{Offset}$ 处，其中 Offset 是在编译时确定的常量。

为了确保影子内存区域始终可用， Offset 的选择必须保证在影子内存分配之前，内存区域 $[\text{Offset}, \text{Offset} + \text{Max_Addr} / 8]$ 未被占用。

影子字节的值编码了对应的八个应用程序字节的可访问状态。值为 0 表示这八个字节均可访问；值为 k （范围为 1 到 7）表示只有前 k 个字节可访问；负值表示这八个字节均不可访问。不同的负值代表不同类型的不可访问内存（如堆越界、栈越界、全局变量越界、已释放的内存等）。

除了使用影子内存储存元数据来为内存检测提供支持，另一种方法是使用内嵌中毒（in-band poisoning）技术，即用一种非常规的中毒值（poison value）初始化红区，并在内存加载和存储操作时检查是否访问了这些被中毒的区域。当然，如果程序合法使用了中毒值，可能会导致误报。对于这种误报，检测工具可以立即使用较慢的检查（例如基于影子内存的检查 [10]）来筛选，或者留待后续的单独验证步骤处理 [3]。

通常情况下，无论哪种方法，都可以通过在释放堆对象时将其标记为红区并配合一个隔离区（quarantine）来提供时间安全性。这种隔离区会延迟将标记为红区的内存重新分配给新的对象，从而进一步增强安全性，降低非法内存重用的概率。

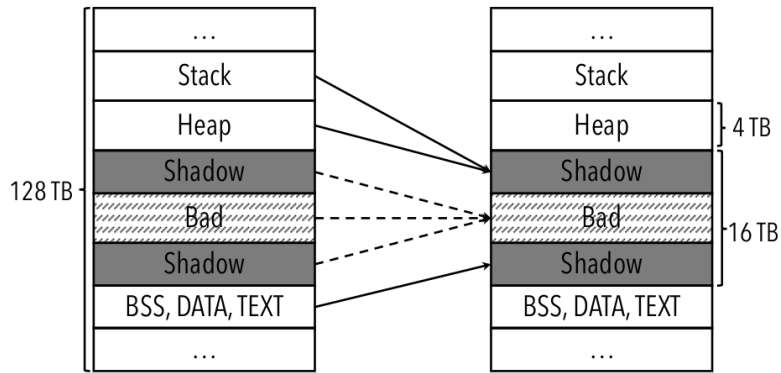


图 2. 在 64 位应用上 ASan 的影子内存分布

2.2 浮点异常

IEEE-754 [11] 标准定义了五种可能引发浮点 (FP) 操作异常的情况：无效操作、除以零、上溢、下溢和不精确计算。主流的处理器架构，如 Intel [12]、AMD [1] 和 ARM [2]，都支持同步浮点异常，并允许用户配置自定义的异常处理程序。在浮点运算结果为非正规数，即当结果太小而无法用标准浮点格式表示的时候，下溢异常会被触发。

非正规数是指指数达到浮点数的最小范围（例如，单精度为 2^{-126} ）的数值，其尾数的首位隐含为零（而正规数的首位为 1）。例如，从 $1.0 \cdot 2^{-126}$ 减去 $1.5 \cdot 2^{-126}$ 的结果为 $0.5 \cdot 2^{-126}$ 。由于指数不能低于 2^{-126} ，该结果必须以非正规形式表示，这需要在尾数中添加前导零。通过启用相关配置，下溢异常会被触发以警告非正规表示可能导致的精度损失。

默认情况下，浮点异常是禁用的。用户可以通过设置浮点控制寄存器（例如，在 x86-64 架构中的 MXCSR）来手动启用这些异常。然而为了优化性能 [1]，x86-64 架构对 IEEE-754 标准进行了简化。只有启用“置零处理” (FTZ, flush-to-zero) 模式时，下溢异常才会被激活，并且所有下溢结果将直接被置为零。

3 本文方法

3.1 本文方法概述

Floatzone 通过异常驱动的运行时检查机制检测内存违规行为，包括空间和时间违规。如图 3 所示，其核心架构包含两个主要部分：

1. 专用内存分配器，用于在对象前后插入红区并隔离已释放的内存。
2. 在编译阶段插入的检查逻辑，用于动态监控每次内存访问是否触及红区。

由于这些检查伴随每次加载和存储操作执行，因此优化运行效率至关重要。为此，Floatzone 引入了一种基于异常的高效方法，能够快速检测红区的访问问题。传统检查工具 (ASan) 通常依赖比较和分支指令，有时还需要查询影子内存。这种方式虽能实现检测，但会显著增加运行时开销 [24]。更理想的解决方案是采用一种低成本的检查机制，该机制可在不显式使用分支的情况下自动检测对红区的违规访问，并在检测到问题时直接中断程序的执行，同时尽量减少与常规代码对执行资源的竞争。

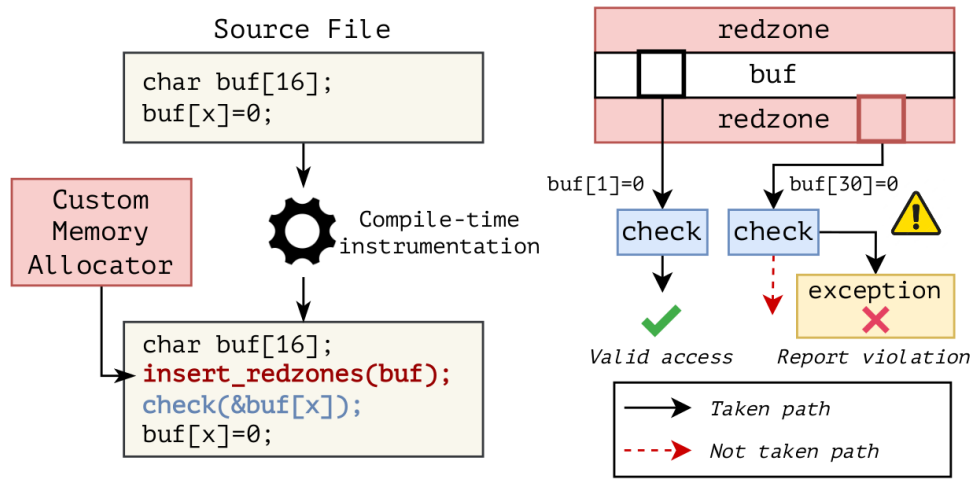


图 3. Floatzone 核心组成和工作流

虽然现代硬件架构未提供专门的内存检查指令，但可以利用浮点指令来实现类似功能。当发生内存违规时，特定的浮点操作能够生成异常。通过精心选择浮点加法操作数，浮点下溢异常被映射为红区的访问检测，从而实现高效而精准的内存违规识别。

3.2 通过浮点异常进行检查

内存检测工具的安全检查结构通常遵循一个固定的两步模式：

1. 判断目标内存位置是否属于红区（比较操作）；
2. 如果检测到违规，分支到相应的错误处理逻辑（分支操作）。

这一模式在现有的检测工具中已成为标准，例如 ASan [21]，其在每次加载和存储操作时都会对对应的影子内存进行比较，并在条件满足时跳转到错误报告函数。然而，这种检查方式会导致约两倍的运行时开销 [21]。近期研究指出，这种开销的主要来源有两个：一是检查本身占据了约 80% [24] 的开销；二是访问影子内存会显著增加页面错误的发生频率。除了这些已知的插桩成本，检测工具还会引入一些微架构层面的性能损耗 [14]：

1. 从影子内存加载数据会驱逐应用程序的缓存行和 TLB 项目；
2. 频繁的比较操作会污染分支预测器缓冲区，并可能导致分支预测失败；
3. 执行单元（如加载、分支和地址生成单元）之间的竞争会引发性能瓶颈。

理想的检测工具应采用一种低成本的指令区分有效和无效的内存访问，这种指令在使用未充分利用的执行单元的同时，也能避免污染缓存和分支预测器缓冲区。尽管在现有硬件上完全避免这些问题并不可能，但浮点运算可以较好地模拟这种理想的检查方式，其核心思路就是将内存安全违规映射到浮点异常。

浮点异常可以看作在错误发生时重定向控制流的条件分支。在所有可用的异常类型中，浮点异常因其灵活性最适合用于表示无效的内存访问。通过精心选择操作数，可以实现仅在操作数等于特定常量值时触发异常。因此论文引入了一种新的检测机制，其形式为 `fp_operation`

(mem[addr], const_value)。该机制通过浮点操作实现比较，用于判断地址 addr 处的值是否等于某个常量值，并在相等时触发异常。该机制将比较封装为浮点操作之后具有三大优势：

1. 基于异常的检查通过 CPU 内部的快速隐式分支验证异常是否发生，从而减少了显式分支的开销；
2. 由于浮点运算不依赖分支预测资源，避免了对微架构缓冲区的污染以及代价高昂的分支预测失败，尽管触发浮点异常比执行分支指令稍慢，但这种慢路径的执行频率非常低；
3. 对于通常未充分利用浮点单元（FPU）的程序，这种检测方法能够提升指令级并行度，从而提高整体性能。

3.3 选取触发浮点异常检查的最佳方案

要找到最适合的浮点运算配置，浮点操作的性能是一个关键方面。因此吞吐量较低的浮点指令（例如除法）首先是被排除的，最优的方案仅限于加法和减法操作，这两者由于符号位的存在而具有等效性。接下来，浮点相加或相减时产生异常的一对值也需要满足某些约束条件，待寻找的这一对值需要尽可能避免与其他数字冲突，以减少误判的可能性。论文指出理想状态是找到一个固定值 x ，使得仅存在一个（或少量）值 y ，当 $x + y$ 时会触发异常，并且 y 需要具有字节级的重复模式（例如，0x4a4a4a4a），这一特性是为了内存对齐的原因。

在考虑上述所有约束条件的情况下，通过暴力搜索浮点数空间，论文找到了一种合适的配置。当 $x = 5.375081 \times 10^{-32}$ （即 $x = 0x0b8b8b8a$ ）时， $x + y$ 仅在 $y = -5.3750813 \times 10^{-32}$ 或 $y = -5.37508 \cdot 10^{-32}$ （即 $y = 0x8b8b8b8b$ 或 $y = 0x8b8b8b89$ ）时会引发下溢异常。值得注意的是， $y = 0x8b8b8b8b$ 具有字节级重复模式。这组特定的数字组合使得 Floatzone 能够通过计算 $\text{float}(y) + \text{float}(0x0b8b8b8a)$ 来表达图 4 中的比较。值得注意的是，仅当 y 等于上述两个已知常量值之一时才会引发异常。

```
1  if( y == 0x8b8b8b8b || y == 0x8b8b8b89 ) {  
2      goto exception_handler;  
3  }
```

图 4. 伪代码表示由于执行 $\text{float}(y) + \text{float}(0x0b8b8b8a)$ 操作所产生的隐式检查

图 5 可视化了下溢的分布来说明。通过固定 x 并尝试所有可能的 32 位 y 值，统计由于计算 $x + y$ 触发的下溢异常次数。可以观察到当 x 值接近 0 时，下溢异常的发生频率更高，而当 x 的绝对值增大时，下溢的频率减少。这是因为仅当 $|x + y| < 1.0 \times 2^{-126}$ 时才会发生下溢异常。换句话说， x 和 y 之间的差值必须足够小，以至于只能用非正规化表示来表示结果。而当 x 值较大时，浮点数的精度不足以生成如此小的结果。

从图 5 的子图中还可以看出，有一组值是下溢发生次数非常少的“最佳值”，并且是以重复模式的数字（0x8b8b8b8b）的形式出现，满足所有的约束要求。

3.4 Floatzone 红区工作原理

红区 (redzone) 是一种有效的技术，用于检测内存错误。最显著的例子是 AddressSanitizer (ASan) [21]，它通过 1:8 字节压缩的影子内存 (shadow memory) 实现红区，其中影子字节

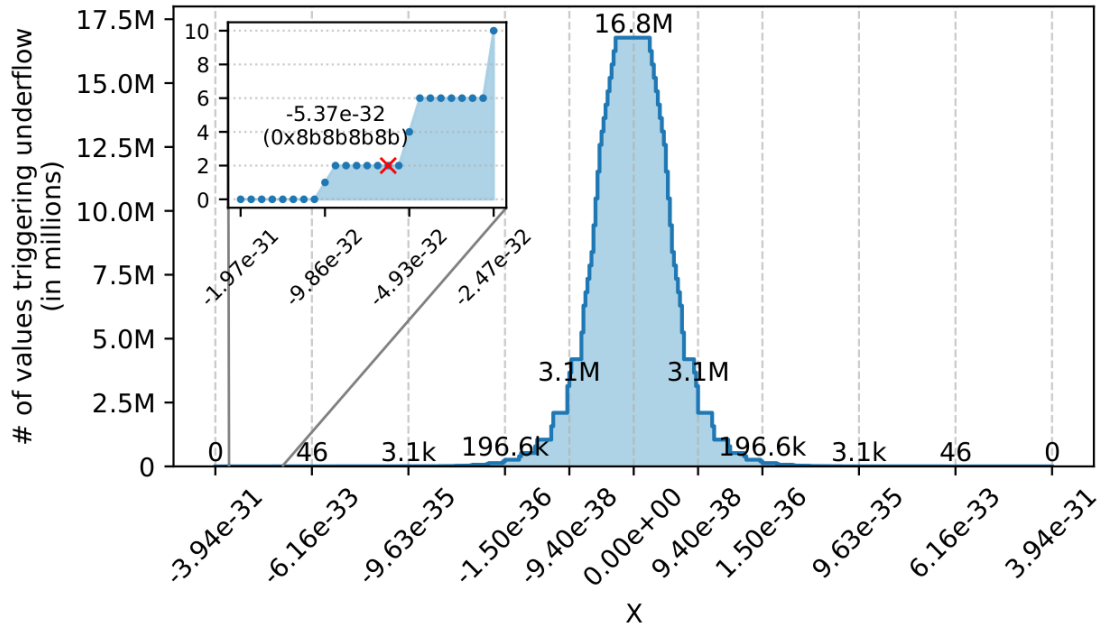


图 5. 通过执行 $x + y$ 得到的下溢异常可视化分布，其中 x 是固定的（ x 轴），而 y 属于单精度浮点数的 32 位空间。放大的子图中突出显示了重复数字 0x8b8b8b8b，它只能与两个其他值相加才会下溢

表示内存是否有效。然而，最近的研究表明，影子内存会显著降低性能，原因包括页面错误（page faults）的增加以及 TLB 和缓存未命中的频率提升 [3, 24]。为避免这些开销，一种更快的替代方案是取消影子内存，只是通过特殊的标记值（poison value）在内存内部（in-band）标记红区 [10]，并在任何内存操作访问被标记的地址时触发警报，而这正是 Floatzone 采取的实现方式。尽管程序在合法使用该标记值时可能会产生误报，但这种情况很少见，可以通过慢检查（使用影子内存）将这些情况排除。

除了误报之外，填充（padding）和对齐（alignment）也是红区解决方案面临的另一个挑战。对齐是必要的，因为即使一个指针并未指向标记值的起始位置，只要其解引用的访问（即使是部分访问）与红区重叠，也应该触发警报 [3]。现有的解决方案通过显式地将检查目标对齐到标记值大小的倍数来实现这一点，但这会因为必须的加法和取模操作而带来额外的开销。此外，当对象未结束在标记值大小的倍数边界上时，就会发生填充问题。检测对填充的越界访问需要更多的检测机制。例如 ReZZan [3] 需要多个加法和减法操作以及两个取模操作来完成检查。

Floatzone 采用了类似的设计，不仅去除了比较和分支操作，还通过精心选择一个对对齐和填充不敏感的标记值 (0x8b8b8b8b)，规避了这些挑战。具体来说，Floatzone 将一个四字节的浮点常量放置在每个内存对象周围，根据需要重复标记值，并用浮点加法对所有内存访问进行检测。如果内存访问操作在标记值上，其对应的加法会导致下溢异常。

通过直接在目标操作的内存地址上执行检测，Floatzone 避免了由于访问影子内存带来的空间局部性代价。为避免填充和对齐问题，Floatzone 使用了一个重复的标记模式 (0x8b8b8b8b)，允许从任何内存访问的起始点读取四字节数据而无需对齐。图 6 直观地展示了这种设计，在红区内的任何四字节访问都会读取到相同的标记模式。

在内存对象销毁时对红区的管理在栈和堆中有所不同如图 7 所示。

对于栈对象，当其生命周期结束（即超出作用域）时，Floatzone 会清除其红区，以避免

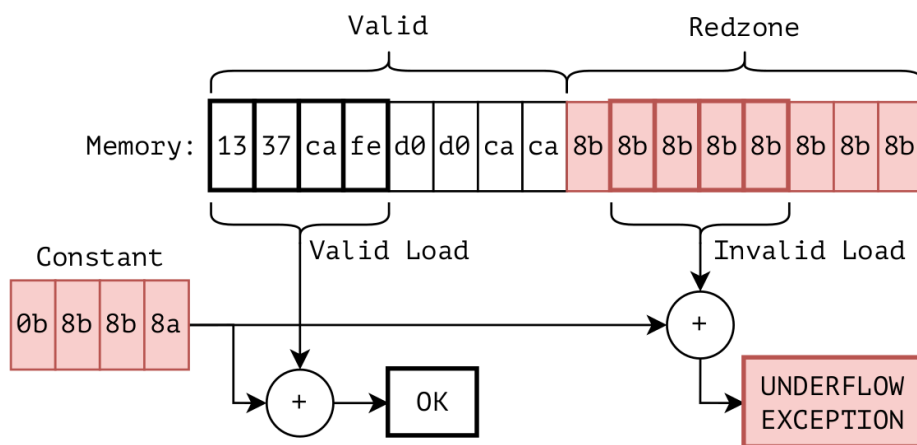


图 6. Floatzone 红区工作细节

在稍后（例如初始化新变量时）遇到遗留的红区字节。

对于堆对象的释放，Floatzone 会将整个对象标记为“有毒”，以检测时间相关的内存错误。为了防止堆内存区域在后续分配中被重新初始化，Floatzone 引入了堆隔离区（heap quarantine），这是一种常见技术，用于检测时间相关的内存错误，它会延迟对象的实际释放。其中堆隔离区采用最近最少使用（LRU）替换策略，当隔离区满（默认 256 MB）时开始释放对象。只要确保对象（包括其红区）在离开隔离区时被清零，就可以避免堆上出现遗留的红区字节。

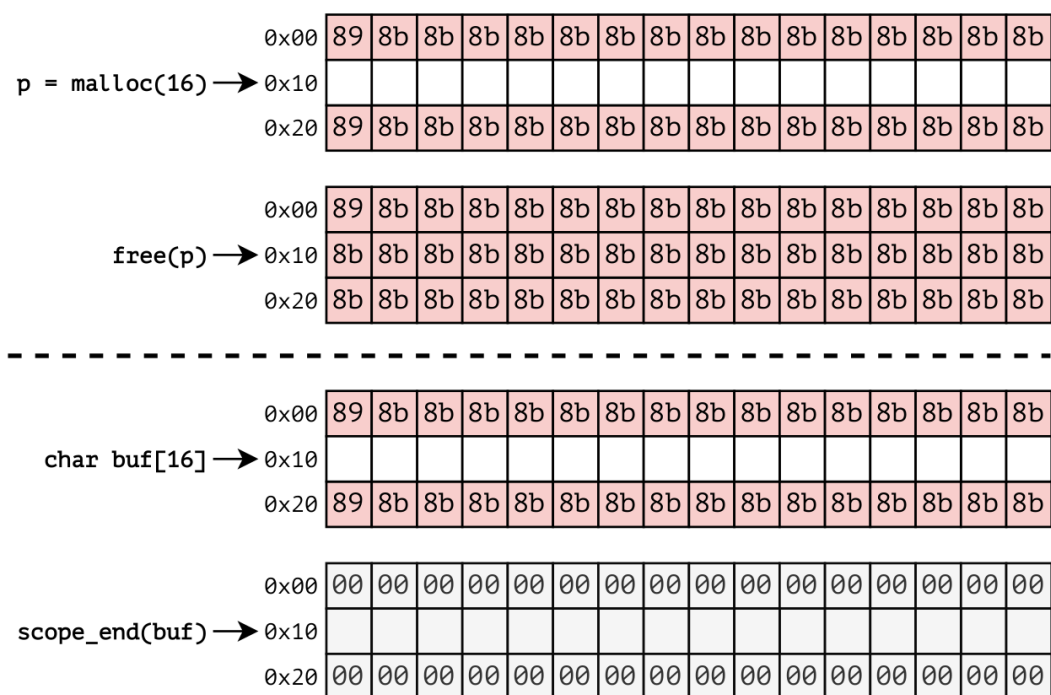


图 7. Floatzone 在堆区和栈区红区管理的不同

在发现重复标记值（0x8b8b8b8b）的时候，同时存在一个额外的冲突值：0x8b8b8b89（最后一个字节是 0x89 而非 0x8b）。尽管这看似一个缺点，但是由于在小端表示中 0x89 是存储的第一个字节，所以 0x8b8b8b89 可以作为红区的起始标记，从而有助于减少误报。

假设图 6 中的有效内存以 0x8b8b 而非 0xcaca 结束，并且程序执行了一个两字节的访问

以读取该值。经过编译之后浮点加法检测代码会插桩在从访问起点开始的四字节值上，此时读取到的值为 `0x8b8b8b8b`（前半部分有效，后半部分属于红区）。因此，这种有效访问会导致意外的异常。然而当使用 `0x89` 作为红区的第一个字节时，这种模式被打破：`0x8b8b898b` 不会触发异常。总而言之，引入 `0x89` 字节可以将以 `0x8b` 结尾的对象与红区的起始部分分开，从而避免误报。

3.5 误报

完全依赖内存内标记的红区的主要缺点是可能会出现误报，因为如果一个程序恰好在内存中使用了配置的标记值（例如 `0x8b8b8b8b`），就会导致意外的异常。Floatzone 在以下情况下能够从这些异常中恢复：如果四字节的标记值并未成为一个完整的红区（即，一个 `0x89` 字节后跟至少 15 个 `0x8b` 字节）。因此，误报仅在以下两种情况下发生：

1. 程序中包含一个完整的红区（但不是由插桩生成的）；
2. 程序中包含一个以 `0x8b8b8b8b` 开头的内存对象。

需要注意的是，在第二种情况下由于无法标记红区的结束位置，因此无法区分该内存对象与它前面的红区，这会导致红区和对象起始部分的实际合并。

最近的研究表明 [3]，不依赖后端真实数据（即没有影子内存）的内存检测器可以同时实现高准确性和性能。论文认为在软件测试中，可能需要处理误报并不是一个问题，因为可以通过其他后端或系统（如 ASan）确认测试用例。尽管存在检测误报的替代方案，例如基于树结构的影子内存，但论文的评估表明，误报的发生非常少，以至于通过异步方式（离线确认）来验证它们已经足够。

3.6 异常处理

由于浮点异常默认情况下是禁用的，所以 Floatzone 在程序启动时启用每个进程的下溢 SIGFPE 异常，同时启用 flush-to-zero（在 x86 上是必需的）。通过注册一个异常处理程序，在每次浮点下溢时，经过插桩的程序都会进入注册了的异常处理程序。这类事件可能有两种情况：

1. Floatzone 插桩的 `vaddss` 指令发生下溢
2. 目标程序中存在的任何通用浮点下溢操作

对于每一个内存访问违规，Floatzone 都会去确认 SIGFPE 是由于插桩引起的。为此恢复对应于触发异常的故障地址是不可避免的，然而 SIGFPE 异常在异常上下文中没有提供故障地址（与 SIGSEGV 不同）。幸运的是，由于 Floatzone 插桩指令中总是包含一个带内存操作数的单一 `vaddss` 指令，因此通过反汇编触发故障的指令来恢复内存操作数的位置是可行的方案。一旦获得故障地址，Floatzone 就可以通过扫描红区来过滤掉误报（例如，确认 `0x89` 红区起始标记是否存在）。

对于偶尔的 SIGFPE 触发进入异常处理程序，Floatzone 会将执行恢复到中断点，因为程序默认会终止。在恢复过程中，简单地忽略异常是不够的，因为恢复到中断点后，由于相关寄存器没有改变，重新执行程序会再次发生故障。因此，Floatzone 通过暂时禁用 flush-to-zero

模式来重新执行故障指令，以计算原本预期的结果，然后将指令指针移动到下一条指令并继续执行。而此方式还避免了 flush-to-zero 意外改变操作结果的问题。例如，将一个小的除数冲刷为零会导致意外的除零错误。

3.7 Floatzone 实现

作为一个编译时插桩工具，Floatzone 的原型基于 LLVM 14 实现。此外 Floatzone 重载了默认的堆内存分配函数以及 C 标准库中常用的 `mem*` 和 `str*` 函数，以便插入内存检查。对于异常处理程序，Floatzone 使用 Intel XED 库 [13] 在运行时反汇编指令。

为了确保插桩的正常工作，有一些特殊情况需要考虑。其中最重要的是堆栈上的红区在关联对象的生命周期结束后必须被清理，否则在检查检查阶段可能会遇到“幽灵”（即旧残留的）红区字节，从而导致误报。大多数这种情况已经在之前的研究中描述过 [6]，例如 `setjmp/longjmp` 和 (C++) 异常。由于 Floatzone 在堆栈上保存和恢复处理器状态时，使用了 `XSAVE` 和 `XRSTOR` 指令，而这些指令可能会在 XMM 寄存器中仍然存在红区，最终导致整个红区被保存在堆栈上。但这种行为仅出现在动态符号查找过程中，而 Floatzone 通过使用 `bind-now` 符号解析编译目标二进制文件来解决这一问题。


4 复现细节

4.1 与已有开源代码对比

4.1.1 引用到的代码

本次复现实验参考了论文的开源代码，代码仓库位于 Github 上面，链接为 [Floatzone](#)。Floatzone 的官方仓库只包含了 Floatzone 的构建入口文件，如 `install.sh` 等，仓库主要部分由数个子仓库组成，如图 8 所示，它们分别为：

1. `floatzone-llvm-project`：由于 `floatzone` 是基于 `llvm-14` 来实现的，因此此子仓库包含了 `floatzone` 的核心实现，包括浮点异常检测和红区管理的代码实现
2. `instrumentation-infra`：此子仓库为 `floatzone` 用于性能测试的自动化代码，但是我在使用之后发现此子仓库的测试代码只适用于 `Floatzone`，而且耦合性很高，经过我大量的移植努力之后再使用此子仓库对同行工作进行测试，观察到使用此子仓库的自动化测试代码并不能完全发挥出同行的应有水准，有碍实验的公平性。因此后面我借鉴了此子仓库的部分思路，重新设计了自动化测试代码，并且取得了预期结果。
3. `XED` [13]：此子仓库是一个用于编码和解码 X86 (IA32 和 Intel64) 指令的软件库，用于在发生浮点异常的时候获取上下文信息，并将这些信息报告出来；同时 `XED` 也用作在发生浮点异常之后能够恢复中断点重新执行程序，这对速度性能测试至关重要。
4. `mbuild`：此仓库是一个基于 Python 的构建系统，主要用于构建 X86 Encoder Decoder (XED) 库。`mbuild` 的设计目标是提供一个跨平台的构建工具，用于处理 `XED` 的代码生成和编译过程，简化在不同操作系统（如 Linux、Windows 和 macOS）上的构建过程。它通过 Python 脚本实现，减少了对其他外部构建工具的依赖。


floatzone
Public
Watch 17

main
2 Branches
2 Tags

Add file
Code

User
Revert "gperftools"
9ec7024 · 7 months ago
20 Commits

example	Added UAF example	2 years ago
floatzone-llvm-project @ 0fa3f55	Fixed install script by unsetting FLOATZONE_MODE	2 years ago
instrumentation-infra @ 5bfbf68	Added infra submodule	2 years ago
mbuild @ 75cb46e	First install script. WIP not working yet	2 years ago
runtime	fix stack clear	2 years ago
xed @ 9fc12ab	Added xed submodule	2 years ago
.gitignore	Minor cleanup	2 years ago
.gitmodules	Added infra submodule	2 years ago
README.md	Added Juliet	2 years ago
env.sh	Finalized README	2 years ago
install.sh	Small typo on install.sh	2 years ago
run.py	Added Juliet	2 years ago

图 8. Floatzone 官方主仓库

4.1.2 改进一：对统一测试脚本从而保证公平性

实验的公平性对于一个工作的评估起着至关重要的作用，其中测试工具的统一起着关键作用。Floatzone 的测试脚本位于仓库 [instrumentation-infra](#)。虽然这个测试脚本看起来非常全面，但是在实际使用的时候遇到了下面几个痛点：

1. 使用相同的 Floatzone 的测试脚本来测试同行工作的时候，我观察到其并不能充分反应同行工作的应有水准，数据有失真的迹象；
2. Floatzone 的测试脚本代码量庞大，而且耦合严重，难以修改其中的测试逻辑，并且难以排除其中对 Floatzone 本身是否有额外的加成；
3. Floatzone 的测试脚本运行所需要的参数非常多，虽然说可配置性看起来高，但是上手成本高，排查问题难度大，在团队开发时不应花费过多时间研究如何使用测试脚本，而是应当易于使用通过实验尽快拿到 baseline；

```
$ python3 run.py \  
    report spec2006 \  
    results/run.2023-06-20.15-37-32/ \  
    --aggregate geomean \  
    --field runtime:median \  
    maxrss:median
```

4. SPEC CPU 由若干个子项目组成，启动 SPEC CPU 测试基准的命令为 runspec，一般运行 runspec 命令之后会将所有项目的编译运行日志文件输出到 SPEC CPU 的指定结果目录下的一个详细日志文件中，便于整体查看与分析。而 Floatzone 的测试脚本将每个 SPEC CPU 子项目拆开单独运行并且分别对每个子项目进行非统一的配置，导致不仅每个子项目的配置是否有利于 Floatzone 无法确认，而且日志文件多且零散难以分析。

针对以上痛点，我重写了 SPEC CPU 2006 测试脚本，其具有以下几个特点，切实地解决了我们团队的需求：

1. 脚本使用 bash 编写，无其他依赖库而且跨平台兼容性强；
2. 脚本内置了统一的 SPEC CPU 2006 的配置，无论在什么平台上面测试都可以保证配置的一致性；
3. 脚本运行参数非常少，只有三个，分别是每次运行的标签，测试数据集的大小，测试的项目，简单明了；

```
$ ./run_spec_clang_asan.sh TAG size benchmark
```


4. 在脚本三个运行参数里面，第一个参数也就是每次运行的标签有着非常重要的作用。每次运行的标签应该有所不同，SPEC 会为每个标签创建一个独立的编译运行目录，而 specmake 会根据 SPEC 的源代码有没有改变进行增量编译，如果本次运行的标签与上一次相同，并且只是修改了编译器代码，那么在编译的时候编译器的变动部分将不能反应到基准测试上面，增量编译时直接跳过编译项目，导致测试结果失真；
5. 正确了解使用 SPEC CPU 2006 并非易事，以下是我在编写脚本时的思考与改进：
 - (a) 由于我研究的方向是关于 C/C++ 漏洞检测的，并不需要评估包含 Fortran 代码的 SPEC CPU 子项目，在运行基准测试的时候需要将 Fortran 编译器 FC 设置为无效指令（例如 echo）避免影响分析；
 - (b) Address Santizer 默认情况下在检测到内存漏洞时会终止程序，导致程序提前退出，从而使得性能测试不准确。为了解决这个问题，我们需要指定 ASAN_OPTION 使得 ASan 在检测到内存漏洞的时候可以恢复现场重新运行，这个不仅需要指定 halt_on_error=0，也需要指定 -fsanitize-recover=address；
 - (c) 在不同的系统平台上面运行 SPEC CPU 2006 项目时，需要在指定编译的 C/C++ 标准，编译 C 语言程序需要使用 -std=gnu89，编译 C++ 程序需要使用 -std=c++98，如果不指定标准 SPEC CPU 2006 的大部分项目都无法正常编译；
 - (d) 由于 SPEC CPU 2006 的每个项目需要的额外参数有所区别，于是我调研了大量关于 SPEC CPU 2006 的技术文章，找出了一套合适的参数配置，将其写进了 SPEC 的统一配置里面，保证了实验的公平性；

```
$ cat << EOF > config/$name.cfg
400.perlbench=default=default=default:
CPORTABILITY= -DSPEC_CPU_LINUX_X64
403.gcc=default=default=default:
CPORTABILITY    = -DSPEC_CPU_LINUX
462.libquantum=default=default=default:
CPORTABILITY= -DSPEC_CPU_LINUX
464.h264ref=default=default=default:
CPORTABILITY    = -fsigned-char
481.wrf=default=default=default:
wrf_data_header_size = 8
CPORTABILITY    = -DSPEC_CPU_CASE_FLAG -DSPEC_CPU_LINUX
482.sphinx3=default=default=default:
EOF
```

6. 我把在探索正确运行 SPEC 过程中遇到的问题以及排除问题的思路写成了[博客](#)。

4.1.3 改进二：解除 Floatzone 只对特定二进制代码启用检测的限制

在使用 Floatzone 时，我做了大量实验归纳出了一个规律，同一份漏洞代码，只有当用 Floatzone 编译出的可执行文件的文件名以 CWE 或者以 run_base 开头才会触发 Floatzone 的漏洞检测。

```
root@floatzone:~/floatzone/example# $FLOATZONE_C buggy.c
root@floatzone:~/floatzone/example# ./a.out
root@floatzone:~/floatzone/example# mv a.out run_base
root@floatzone:~/floatzone/example# ./run_base
Exception caught: fault_addr: 0x7fff305f8788
!!!! [FLOATZONE] Fault addr = 0x7fff305f8788 !!!!
0x7fff305f8748: 00 00 00 00
0x7fff305f874c: 00 00 00 00
```

我观察漏洞代码编译成 llvm 的中间语言 ir，发现经过 Floatzone 的编译在源程序中确实插入了浮点检测指令 vaddss。

```
root@floatzone:~/floatzone/example# cat buggy.ll | grep vaddss
vaddss (%rcx), %xmm0, %xmm15
vaddss (%rcx), %xmm0, %xmm15
```

换言之，Floatzone 的确插入了检测代码，但是没有启用。通过全局搜索关键词 run_base，我找到了触发错误的关键代码，也就是在主仓库的 runtime 目录下，在图 8 中可以看到此文件夹。阅读 runtime 目录下的 wrap.c 代码，我发现确实存在启用漏洞检测的开关。

```
root@floatzone:~/floatzone/runtime# cat wrap.c
...
#define TARGET "run_base" // use "run_base" for SPEC
#define JULIET "CWE"      // use "CWE" for Juliet
...
if(strstr(ubp_av[0], TARGET) || strstr(ubp_av[0], JULIET)){
    // register signal handler
    struct sigaction action;
    memset(&action, 0, sizeof(struct sigaction));
    action.sa_flags = SA_SIGINFO;
    action.sa_sigaction = handler;
    ...
}
```

将上述代码的平台特定判断语句去除之后就可以解除 Floatzone 对特定二进制代码的限制。

4.1.4 改进三：在 Floatonze 检测到错误之后不提前终止程序

在运行经过 Floatzone 编译的漏洞程序时，即使漏洞程序包含多个漏洞，但是漏洞程序依旧会在第一个漏洞处终止程序。此现象非常不利于 SPEC CPU 进行性能测试，因为检测到漏洞而提前终止会导致基准运行时间更短，有碍实验公平性。经过验证，Floatzone 并没有类似于 ASan 的 `-fsanitize-recover=address` 指令，需要修改运行时库重新编译。在 `runtime` 目录中，我找到了相关代码：

```
root@floatzone:~/floatzone/runtime# cat wrap.c
...
    #if FUZZ_MODE == 1
        abort();
    #else
        exit(FAULT_ERROR_CODE);
    #endif
...
```

这里我们可以观察到无论是否启用模糊测试的宏，程序都会异常终止，于是把这一段异常退出代码去除之后再将其用于 SPEC 基准测试，最终达到预期效果。

4.1.5 改进四：增强边界检查提高 Floatzone 准确性

4.2 实验环境搭建

1. 如图 9 所示，实验机器的配置为英特尔 Xeon E5-2670，12 核 24 线程，64G 内存和 515G 硬盘，系统为 Debian 系统，由于 FLoatzone 需要使用英特尔的指令拓展，所以物理上只能使用英特尔的 x86 CPU；
2. 本次复现涉及到的四个相关工作都在宿主机上的容器环境运行；
3. 对于 Floatzone，在宿主机里运行以下命令搭建实验环境：

```
$ docker run -it --hostname floatzone --name floatzone \
    ubuntu:22.04 /bin/bash
$ sudo apt install \
    ninja-build \
    cmake gcc-9 \
    autoconf2.69 \
    bison build-essential \
    flex \
    texinfo \
    libtool \
    zlib1g-dev
```

```
root@szu-poweredger730xd:~# neofetch
_,met$$$$$gg.      root@szu-poweredger730xd
,g$$$$$$$$$$$$$P.  -----
,g$P$P"            ""Y$$.
,$$P'              `$$$
',$$P              ,ggs.  `$$b:
`d$$'              ,P"'   $$$
$$P                d$'    ,  $$P
$$:                $$    -   ,d$$'
$$;                Y$b._   _,$P'
Y$$.              `."Y$$$$P""
`$$b              "-._
`Y$$
`Y$$.
`$$b.
`Y$$b.
`"Y$b._
`""

OS: Debian GNU/Linux 12 (bookworm) x86_64
Host: PowerEdge R730xd
Kernel: 6.1.0-26-amd64
Uptime: 47 days, 3 hours, 37 mins
Packages: 2713 (dpkg)
Shell: bash 5.2.15
Resolution: 1680x1050
Terminal: /dev/pts/2
CPU: Intel Xeon E5-2670 v3 (24) @ 3.100GHz
GPU: 0a:00.0 Matrox Electronics Systems Ltd. G200eR2
Memory: 8959MiB / 64301MiB
```

图 9. 实验的宿主机配置

```
$ pip3 install psutil terminaltables
$ cd ~
$ git clone https://github.com/vusec/floatzone.git --recurse-submodules
$ cd floatzone
$ source env.sh
$ ./install.sh
```

4. 对于 ASan, 在宿主机里运行以下命令搭建实验环境:

```
$ docker run -it --hostname asan --name asan \
    ubuntu:latest /bin/bash
$ sudo apt install clang build-essential
```

5. 对于 Giantsan, 在宿主机里运行以下命令搭建实验环境, 由于 Giantsan 只是开源了可执行文件, 而没有开源源代码, 所以直接把仓库克隆下来就可以使用:

```
$ docker run -it --hostname giantsan --name giantsan \
    ubuntu:20.04 /bin/bash
$ cd ~
$ git clone https://github.com/AceSrc/GiantSan-Artifact
```

6. 对于 ASan-, 在宿主机里运行以下命令搭建实验环境:

```
$ docker run -it --hostname asan-- --name asan-- \
    ubuntu:latest /bin/bash
$ git clone https://github.com/junxzm1990/ASAN--.git && cd ASAN--
```



```
$ cd llvm-4.0.0-project
$ mkdir ASan--Build && cd ASan--Build
$ cmake -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -G "Unix Makefiles" ../llvm
$ make -j
```

7. SPEC CPU 2006 作为一个标准测试基准也是需要安装的，在本次复现任务中，我给每个开发容器都安装了一个：

```
$ git clone https://gitee.com/cao_wuhui/spec-cpu-2006-v1.0.1/
$ cd spec-cpu-2006-v1.0.1
$ source shrc
$ ./install.sh
```

8. JTS 测试集 (CWE) 从官方网站 <https://samate.nist.gov/SARD/test-suites/112> 下载之后在顶层目录 make 就可以编译了；
9. CVE 是真实世界里面的漏洞，而 Linux Flaw 的 CVE 漏洞需要下载虚拟机镜像来运行，如图 10 所示，下载的 仓库 开源在 Github 上。

LinuxFlaw

This repo records all the vulnerabilities of linux software I have reproduced in my local workspace.

If the vulnerability has both CVE-ID and EDB-ID, CVE-ID is preferred as its directory name. All the vulnerable source code packages are stored in [source-packages](#)

Vmware Workstation Images

Image Name	username	password
Ubuntu 8.10	exploit	exploit
Ubuntu 10.04LTS	exploit	exploit
CentOS 6.5	core	core
CentOS 5.5	core	core
Ubuntu 11.04	dzm77	dzm77
Ubuntu 12.04	ubuntu	ubuntu
Fedora	fedora	fedora
OpenSUSE	core	core
Ubuntu 14.04_core	core	core
Kali	root	kali
Ubuntu_14.04_alex	research-cve	toortoor
Ubuntu_14.04_pt	pt	pt

图 10. CVE 镜像

4.3 创新点

关于创新点和改进点我已经在 5.12, 5.13, 5.14 小节进行了详细的阐述。

1. 对统一测试脚本从而保证公平性
2. 解除 Floatzone 只对特定二进制代码启用检测的限制
3. 在 Floatzone 检测到错误之后不提前终止程序

5 实验结果分析

5.1 SPEC CPU 2006 性能测试

表 1 展示了 Floatzone、ASan、ASan-、Giantsan 在每个 SPEC CPU2006 运行时倍率，表格里每个子项目的倍率结果等于使用各类工具插桩之后的运行时间，除以插桩前的运行时间。由于这四个工作的 llvm 开发版本不一样，为了排除版本不同带来的性能差异，此处使用倍率去衡量工作的性能是一种公平的做法。这四个工作中，ASan- 是 USENIX 2022 的工作，Floatzone 是 USENIX 2023 的工作，Giantsan 是 USENIX 2024 的工作。平均而言，Floatzone 的几何平均运行时开销为 31%，这与论文中所提到的 Floatzone 34% 的 overhead 非常接近，可以认为是成功复现了论文内容。

对于 429.mcf, 462.libquantum, 470.lbm, 我观察到了负的 overhead, 这是在复现论文中是没有提到的，因为复现的论文最后的附录里面所有的 overhead 都是正的。但是我认为，这些依旧是可以解释的数据，这些负的 overhead 可以认为是性能抖动。在这三个 SPEC 子项目里面，Floatzone 进行浮点检查的次数非常少，所以带来的 overhead 几乎可以不计。从整体分析，每一种内存检查工具在这三个项目中的性能开销都偏小，这也说明了在这三个子项目里面需要进行内存检查的次数非常少。同时论文最后的附录也可以佐证我的观点，虽然说论文附录里面的性能开销都是正的，但是在这三个 SPEC 子项目的性能开销都小于 4%，由此可以得出尽管我的复现结果虽然是负的开销，但是可以认为是性能抖动。

对于 464.h264ref 视频压缩子项目，其开销由于 Floatzone 的浮点检查而显得特别大。对于这一特定子项目，开销主要来自对 libc 的 mem* 系列函数的检查，而现阶段的 Floatzone 总是在调用这些标准库函数之前执行检查，后续这可以通过将检查整合到函数内部进行优化（例如，将检查与 memcpy 循环合并以避免重复循环）。此外，我们观察到栈插桩在所有 SPEC 子项目上相对廉价，而堆 redzone 的成本总体上也较低，但一些分配密集型的二进制文件（如 471.omnetpp 和 483.xalancbmk）由于堆分配器中的额外操作而产生了显著的开销。对于 458.sjeng 项目，Giantsan 未能成功运行，原因是 SPEC 版本不一致导致的，Giantsan 的测试环境是 SPEC CPU 2006 v1.2 版本，而我实验室目前使用的版本是 v1.01 版本。

总体而言，虽然 Floatzone 在性能方面全面领先其他工作，但是 Floatzone 在 SPEC CPU 2006 运行过程中经历了不可避免的误报，其中 464.h264ref 频繁操作 Floatzone 的浮点标记值导致大量误报产生。由此我们可以合理地推测 Floatzone 的浮点标记值通常包含在图像表示中，其中 15 个像素包含相同的值 (0x8b)，前面是略微调整过的像素 (0x89，误报大多是由于 0x8b8b8b8b 出现在内存对象的开头。所幸的是，这些误报很容易被识别，因为它们可以通过异步方式（例如使用 ASan）进行确认，也就是所谓的 double check。

表 1. SPEC CPU2006 基准测试的运行开销比较

SPEC CPU 2006	ASan	ASan+	GiantSan	Floatzone
400.perlbench	4.16	3.57	3.46	1.52
401.bzip2	1.84	1.74	1.40	1.29
403.gcc	2.82	2.73	2.49	1.71
429.mcf	1.41	1.39	1.34	0.96
433.milc	2.00	1.67	2.25	1.14
434.zeusmp	1.83	1.80	1.51	1.20
445.gobmk	1.71	1.52	1.58	1.19
450.soplex	2.40	2.34	1.92	1.68
453.povray	4.65	3.99	4.03	1.68
456.hmmer	2.29	2.43	2.28	1.64
458.sjeng	1.87	1.66	-	1.51
462.libquantum	1.81	1.08	1.59	0.99
464.h264ref	2.54	2.08	1.67	1.94
470.lbm	1.74	1.35	1.19	0.99
473.astar	2.11	2.42	1.85	1.42
482.sphinx3	1.69	1.66	1.93	1.28
483.xalancbmk	11.61	10.51	7.93	2.85
GEOMEAN	2.34	2.13	1.85	1.31

5.2 人造漏洞 CWE 测试集检测准确性分析

NIST Juliet 测试套件 (v1.3) [15] 包含数百个用于检测内存安全错误的测试用例。在此我选择了与空间和时间内存错误相关的错误类别进行复现，这些类别与 ASan- 论文中报告的相同。

表 2. ASan 与 Floatzone 在 Juliet 测试套件上的比较

Description (CWE)	Total	ASan	Floatzone
Stack overflow (121)	72	72	71
Heap overflow (122)	78	74	74
Buffer underwrite (124)	24	24	24
Buffer overread (126)	19	16	19
Buffer underread (127)	24	24	23
Double free (415)	17	17	17
Use-after-free (416)	18	18	18

表 2 说明了 Floatzone 在 Juliet 测试套件中的安全保障与 ASan 几乎相同，并且复现效果与论文相吻合。对于 CWE 122，Floatzone 能检测到比 ASan 多一个堆缓冲区溢出，因为 ASan 没有对 `wmemset` 进行插桩。对于 CWE 126，Floatzone 比 ASan 检测到更多的三个案例，但这些漏洞的影响取决于栈上未定义的数据，因为 CWE 126 存在的一个非空终止 `printf` 表现行为是不确定的，它可能会导致超读或不发生。对于 CWE 127，Floatzone 错过了一个偏移量为 -20 字节（5 个整数下溢）的访问，因为 Floatzone 默认的红区长度为 16 字节，也就是说这个下溢跨过了一个完整红区之后又落入了一个合法区域里面，这种错误在 ASan 中同样无法检测到。总的来说，我们可以观察到 Floatzone 与 ASan 的检测能力相当。

5.3 现实漏洞 CVE 测试集检测准确性分析

同时我也复现了在 Linux Flaw [18] 项目的一些 CVE 上 Floatzone 的漏洞准确性验证，这与 ASan- 和 SANRAZOR 论文中的做法类似。在此我采取和 Flotzone 一样的措施，省略了一些 Floatzone 不关注的漏洞类型，例如栈耗尽和空指针解引用。

表 3 展示了每个 CVE 测试用例的检测结果，其中有两个案例 Floatzone 未能检测到漏洞，本次复现结果和论文实验结果一致。首先是 CVE-2009-2285，这是一个偏移一个字节的下溢，这是 Floatzone 设计的已知限制，因为 Floatzone 只有读取到的四个字节全部是红区字节时才有可能触发错误，而部分越界是无法检测到的，如图 11 所示。其次是 CVE-2017-7263，这是一个在 4 字节对象上偏移 1016 字节的错误。显然，我们的 16 字节溢出 redzone 太小，无法检测到这个漏洞。这显示了嵌入型红区的局限性，尤其是在没有后续对象的情况下，但是 ASan 能够检测到这些野指针访问，因为它们会落入影子内存中的“未映射即为有毒”区域。总的来说，Floatzone 检测到了 18 个实际漏洞中的 16 个。

表 3. ASan 与 Floatzone 在 CVE 测试套件上的比较

CVE	Type	ASan	Floatzone
CVE-2006-2362	stack-buffer-overflow	✓	✓
CVE-2009-1759	stack-buffer-overflow	✓	×
CVE-2009-2285	heap-buffer-overflow	✓	×
CVE-2013-4243	heap-buffer-overflow	✓	✓
CVE-2015-8668	heap-buffer-overflow	✓	✓
CVE-2017-12858	heap-use-after-free	✓	✓
CVE-2015-9101	heap-buffer-overflow	✓	✓
CVE-2016-10095	stack-buffer-overflow	✓	✓
CVE-2016-10270	heap-buffer-overflow	✓	✓
CVE-2016-10269	heap-buffer-overflow	✓	✓
CVE-2017-5976	heap-buffer-overflow	✓	✓
CVE-2017-5977	heap-buffer-overflow	✓	✓
CVE-2017-7263	heap-buffer-overflow	✓	×
CVE-2017-12858	heap-use-after-free	✓	✓
CVE-2017-12937	stack-buffer-overflow	✓	✓
CVE-2017-14407	stack-buffer-overflow	✓	✓
CVE-2017-14408	stack-buffer-overflow	✓	✓
CVE-2017-14409	global-buffer-overflow	✓	✓

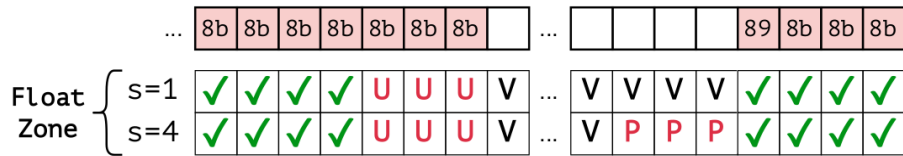


图 11. Floatzone 检测能力的可视化展示。每个单元格表示一个内存访问（即加载/存储）的开始，大小为 s。符号 V 表示有效的（在边界内的）内存访问，✓ 表示真实的正例越界，U 表示下溢误报，P 表示部分上溢误报

6 总结与展望

6.1 总结

本文复现了提出一个名为 Floatzone 内存检测工具的 USENIX 2023 文章，它展示了如何利用浮点运算来实现常见的比较与分支检测器模式，通过精心设计的浮点下溢异常来识别内存违规行为。用于内存安全的检测工具（Sanitizers）已成为软件测试中的标准技术，尽管近年来进行了优化，但最先进的漏洞检测工具仍然带来了显著的运行时开销。而 Floatzone 在普通硬件上引入更快的有效性检查进一步提高了性能。复现结果和论文阐述的内容整体一致，证明了通过浮点加法执行这些检查确实比传统的比较指令更快，这得益于多种微架构级别的优化。最后的复现实验表明，Floatzone 在运行时和内存开销方面（去除了 Shadow Memory）都显著优于目前最先进的技术，并且漏洞检测的能力并没有随着性能的提升而下降。而本次复现也存在不足，由于时间原因，我并不能深入了解模糊测试相关的工作，因此本次复现任务我并没有复现论文中模糊测试的部分。

6.2 展望

1. 未来进一步了解模糊测试的相关知识；
2. Floatzone 的红区 poison 值目前是四个字节重复的模式，如果可以找到八字节重复的 poison 值并且触发 64 位下溢的可能性足够小，则可以进一步较少误报，提高检测性能与准确性；
3. 由于 Floatzone 在用了一条浮点加法指令进行漏洞检测情况下，overhead 只有 31%，如果我用两条浮点加法指令进行检测，一条检测区域头部，一条检测区域尾部，不仅提高了准确性，而且 overhead 里面上限也只是 62%，依旧可以打败 USENIX 2024 的 SOTA Giantsan，我目前正在实现这个想法。

参考文献

- [1] AMD. *AMD64 Architecture Programmer's Manual*. Sunnyvale, California, 2020.
- [2] Arm. *Architecture Reference Manual - Armv8, for A-profile architecture*. Cambridge, UK, 2021.
- [3] Jinsheng Ba, Gregory J Duck, and Abhik Roychoudhury. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
- [4] 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.

- [5] 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.
- [6] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. Stack bounds protection with low fat pointers. In *NDSS*, volume 17, pages 1–15, 2017.
- [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] Istvan Haller, Erik Van Der Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security*, pages 1–6, 2016.
- [9] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.
- [10] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.
- [11] IEEE. Ieee standard for floating-point arithmetic. Technical Report 754-2019, IEEE, New York, August 2019.
- [12] Intel, Santa Clara, California. *Intel® 64 and IA-32 Architectures Software Developer’s Manual combined volumes*, 2019.
- [13] Intel. X86 encoder decoder (xed), 2022.
- [14] Yuseok Jeon, WookHyun Han, Nathan Burow, and Mathias Payer. {FuZZan}: Efficient sanitizer metadata design for fuzzing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 249–263, 2020.
- [15] Frederick Boland Jr. and Paul Black. Juliet 1.1 C/C++ and Java Test Suite. *IEEE Computer*, 45(7):88–90, July 2012.
- [16] 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.
- [17] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1901–1915, 2022.

- [18] Dongliang Mu. Linuxflaw, 2015.
- [19] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2015.
- [20] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.
- [21] 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.
- [22] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.
- [23] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: compositing security mechanisms through diversification. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 271–283, 2017.
- [24] 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.