LLVM Compiler-rt 简介

及其 RISC-V 支持情况调研与工作展望

刘子康（实习）

PLCT-LLVM 小队

2021 年 1 月 19 日

Compiler-rt 项目结构

- LLVM 项目的一部分，大部分由 Google 开发并贡献到 LLVM 项目之中，用于 Android,Chromium 等项目开发。
- builtins: 将目标机器不支持的操作转换为一系列支持的操作 (软件模拟实现)。
- **sanitizers:** Google 推出的一系列运行时工具，以效率和空间为代价换取对内存泄漏等问题的检测。
- profile: Profiling 工具，收集 coverage 信息等，用于 Profile-Guided Optimizations。
- 其他工具 (cfi, safestack, ...): 安全性、内存分配、代码覆盖率、测试、统计数据等。

AddressSanitizer

- 内存错误检测器
- 检测以下错误：
  Use after free
  (Heap | Stack | Global) buffer overflow
  Use after return
  Use after scope
  Initialization order bugs
  Memory Leaks
- 平均减缓运行速度 2x
- 由以下两部分组成
  一个 LLVM
  ModulePass(llvm/lib/Transforms/Instrumentation/)
  Compiler-rt 下的 lib(compiler-rt/lib/asan)

AddressSanitizer：原理简介

- 具体可参考原始论文 *Addresssanitizer: A fast address sanity checker*.
- 使用 Shadow memory 技术，scale+**offset** 方式计算 shadow.
- scale 可自行定义，默认为 3，offset 和 platform 相关。

AddressSanitizer：实现简介

- 具体可参考这篇知乎专栏《工欲善其事必先利其器——AddressSanitizer》[1]
- ModulePass 负责插桩
- lib 提供 Instrumented syscalls(malloc, etc.)
- 不用重新编译 c library, 由编译器劫持 (Intercept) 部分 syscall 到 ASan 的 lib。

[1] https://zhuanlan.zhihu.com/p/382994002

LeakSanitizer

- ASan 的一部分，也可单独使用 (-fsanitize=leak)
- 设计文档:
  `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizerDesignDocument`
- 不插桩，进程生命周期将要结束时暂停执行
  从 root set 开始迭代扫描所有指向堆块的指针，标记这些块为可访问的
  再迭代访问现存的堆块，对比两个集合得出泄露的内存资源

UndefinedBehaviorSanitizer

- 检测未定义行为
- 在可能未定义的指令前做判断检测（如：对 32bit 无符号数的左移位数为 32)
- `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

MemorySanitizer

- 检测 C++ 未初始化的内存使用
- 具体可参考论文 *MemorySanitizer: fast detector of C uninitialized memory use in C++*
- 使用 shadow memory 技术，每 1bit 的内存对应 1bit 的 shadow memory (ASan 是 1byte-1bit)
- 3x 的性能损耗 2x 的内存占用

ThreadSanitizer

- 检测 C/C++ 中的数据竞争
- 5-10x 的内存开销 2-20x 的执行时间开销
- 需要所有依赖的库都被重新编译以确保正确性
- 参考论文 *ThreadSanitizer –data race detection in practice*
- 对 Lamport Lock 的应用
- 参考`https://zhuanlan.zhihu.com/p/38687826`

DataflowSanitizer

- 提供一个通用的 dataflow 分析框架供用户分析自己的程序数据
- 需要 include 相关头文件，自行在代码中插桩
- 需要提供一个插桩函数的列表重新编译 libc++
- `https://clang.llvm.org/docs/DataFlowSanitizer.html`
- 活跃开发中

KASAN, KMSAN, KCSAN

- 内核版本的 Sanitizers
- KMSAN 单独 host[2]
- KASAN 和 KCSAN 在 linux kernel 里
- 可参考如下文档:
  `http://www.wowotech.net/memory_management/424.html`
  `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html` `https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html`

[2] github.com/google/kmsan

CFI, SafeStack, Stats

- CFI: Control Flow Integrity
  `https://clang.llvm.org/docs/ControlFlowIntegrity.html`
- SafeStack: **防止栈缓冲溢出攻击**
  `https://clang.llvm.org/docs/SafeStack.html`
- Stats: **收集 CFI 的使用数据，多年未更新**

Profile

- Profile: 用于 Profile Guided Optimization
- 10 ~20 % 的性能提升
- `https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization`
- `https://llvm.org/docs/HowToBuildWithPGO.html`

Memprof, Scudo, Fuzzer, Xray

- Memprof: 类似 ASan 的办法插桩，记录内存使用情况，只支持 X86
  `https://lists.llvm.org/pipermail/llvm-dev/2020-June/142744.html`
  `https://reviews.llvm.org/D87120`
- Scudo: 内存分配器，相比 ASan 是内存错误的缓解（而非检测）工具.
  `https://source.android.com/devices/tech/debug/scudo`
- Fuzzer: 模糊测试
  `https://llvm.org/docs/LibFuzzer.html`
- XRay: 函数调用跟踪，动态启用禁用 Instrumentation
  `https://llvm.org/docs/XRay.html`

Hardware-assisted AddressSanitizer

- 相比 ASan，利用 64bit 指针的高位来存储内存标签
  类似的 2x CPU 开销和代码大小开销，内存开销大幅降低
- 内存在分配时产生一个随机标签，访问时检测 shadow memory 的值是否相等
- 释放时更改 shadow memory 中的随机值，则原地址再次访问就报错
- 如果某次释放产生了一个某地址用过的标签（标签只有 8 位），则此地址 use-after-free 等不会被检测到
- `clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.htm`
- `source.android.com/devices/tech/debug/hwasan`

RISCV 的相关情况

- 目前 AArch64 和 X86_64 支持 Pointer Masking (或类似的技术)
- 支持 Pointer Masking 的 J 扩展在讨论之中
- `https://github.com/riscv/riscv-j-extension`

ASan 支持 RISCV 做了些什么

- 添加相关的宏定义
- 在 sanitizer_common 中：
  根据 RISCV ABI 添加 Syscall 的格式
  为 RISCV64+Linux 添加 syscall 的 interceptor，如 vfork 和 clone（用于 lsan）
  更新对应常量，如 Thread Control Block Head 的长度
  计算 ThreadAddr[3]
  在 StackTrace 中加入对下条指令地址的计算、堆栈帧地址计算
- 在 asan 中：
  添加 Mapping 参数
  更新 Asan PASS 中的参数 (ShadowOffset)

[3] github.com/riscv-non-isa/riscv-elf-psabi-doc/issues/53

其他的 Sanitizers 支持还需要做些什么

不涉及 Sanitizers 算法，只和底层的 os-specific, platform-specific 有关。
已经实现的共有的调用 Interception，不需要再实现。

- MSAN：类似 ASAN，参考 RISCV 相关规范定义 offset 等
- TSAN：大量需要根据 platform 做的修改，如计算 stack pointer mangling 后的原始值
- DFSAN：platform-specific 的内容少，但其整体复杂度高

整体工作的情况

目前关于 RISC-V 的工作有一个或多个特点：

- 实现简单（不是 target-specific）；标记支持即可。
- 有使用需求；ASan 使用较广，相比之下 MSan 拖慢程序很多，基本只有 Google 自己在用。

我们可以完成的工作

- RISCV32 支持相关：目前上游没有 RISCV32 的支持，没有做的原因是当时 RISCV32 的 linux ABI 没有被声明 stable。目前参考 psABI[4]文档，相关 ABI 已经 frozen。
- 更多的 Sanitizers 支持：tsan, msan, dfsan 等没有相关工作，工作量取决于是否 target-specific, os-specific 等。
  HW-Asan 目前没办法做，因为 J Extension 暂未 Ratified。
- 其他工具支持：关注度低，实现不都复杂，可以在实现并验证确实可用后向上游交 Patch 标记支持。
  可参考如 MIPS 平台有没有支持来决定做不做 RISCV 的支持。
  和 ASan 类似的 Valgrind 参考 PLCT 的 Roadmap.

[4] https://github.com/riscv-non-isa/riscv-elf-psabi-doc

具体情况表

项目	状态
ASan	RISCV64
HWASan	暂无法实现
LSan	RISCV64
Msan	未实现，部分 platform-specific
TSan	未实现，C++ 和 GO 分开，platform-specific 内容多
UBSan	RISCV64
DFSan	未实现，部分 platform-specific

具体情况表-续

项目	状态
CFI	有未合并相关工作，os-specific，需要验证
Fuzzer	os-specific，复杂，工作量大
MemProf	支持架构少，工作量大
Profile	os-specific，需要验证
SafeStack	可即实现，os-specific，需要验证
Scudo	未实现，部分 platform-specific
Stats	RISCV64
Xray	未实现，platform-specific 内容多

其他如 GWPAsan（Android 用），BlockRuntime (Apple 相关）等不在此表列出。

1. Compiler-rt 的官方文档 (资料不全)
   https://compiler-rt.llvm.org/
2. Compiler-rt 的介绍性文档 (部分过时)
   https://www.jianshu.com/p/4f22bfd1a93d
3. Sanitizers 的 Repo
   https://github.com/google/sanitizers
4. DFSan 的文档
   https://clang.llvm.org/docs/DataFlowSanitizer.html
5. HWASan 的设计文档
   https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html
6. RISCV psABI 文档
   https://github.com/riscv-non-isa/riscv-elf-psabi-doc
7. RISCV J Extension
   https://github.com/riscv/riscv-j-extension
8. RISCV Virtual Memory
   https://github.com/riscv/virtual-memory
9. ARM TBI 和 MTE 介绍及部分相关文档
   https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/
   https://developer.arm.com/documentation/den0024/a/ch12s05s01
   https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety
10. ASan+RiscV 的邮件列表（包含 Patches 链接）
    https://groups.google.com/a/groups.riscv.org/g/sw-dev/c/q7FHFkM68qI/m/D75EVIe5AAAJ
11. valgrind 支持 riscv64 的 fork，开发中
    https://github.com/petrpavlu/valgrind-riscv64

相关的 llvm patches:

1. https://reviews.llvm.org/D96954
2. https://reviews.llvm.org/D92403
3. https://reviews.llvm.org/D92464
4. https://reviews.llvm.org/D106888
5. https://reviews.llvm.org/D90574
6. https://reviews.llvm.org/D75168
7. https://reviews.llvm.org/D106919

Thanks!

本次分享的内容已尽力保证时效。
如有错误、疏漏或过时信息，烦请批评指正。