Demystifying Compiler-rt Sanitizers for multiple architectures

Demystifying Compiler-rt Sanitizers for multiple architectures

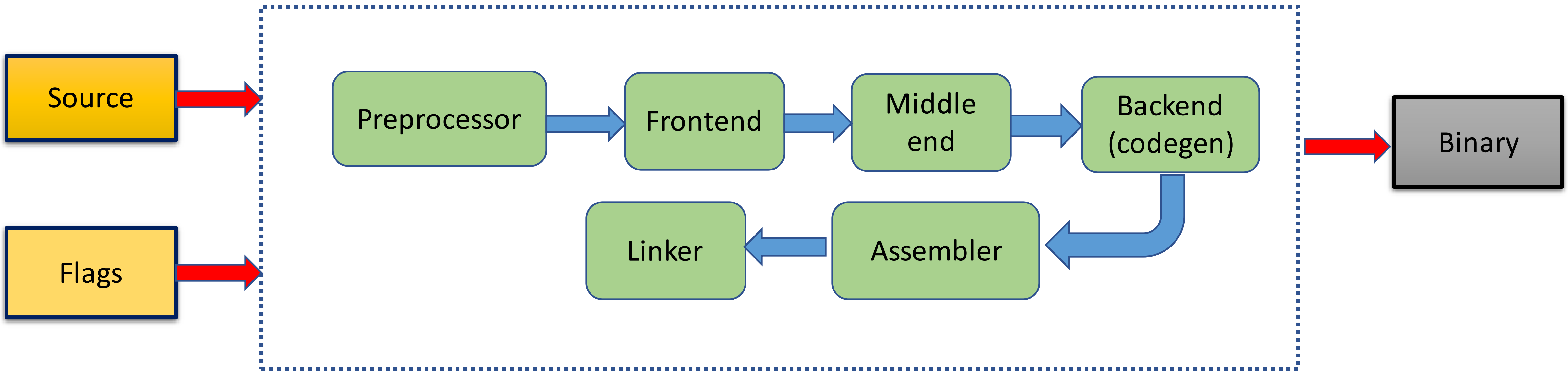
Mamta Shukla, Software Engineer

Leica Geosystems

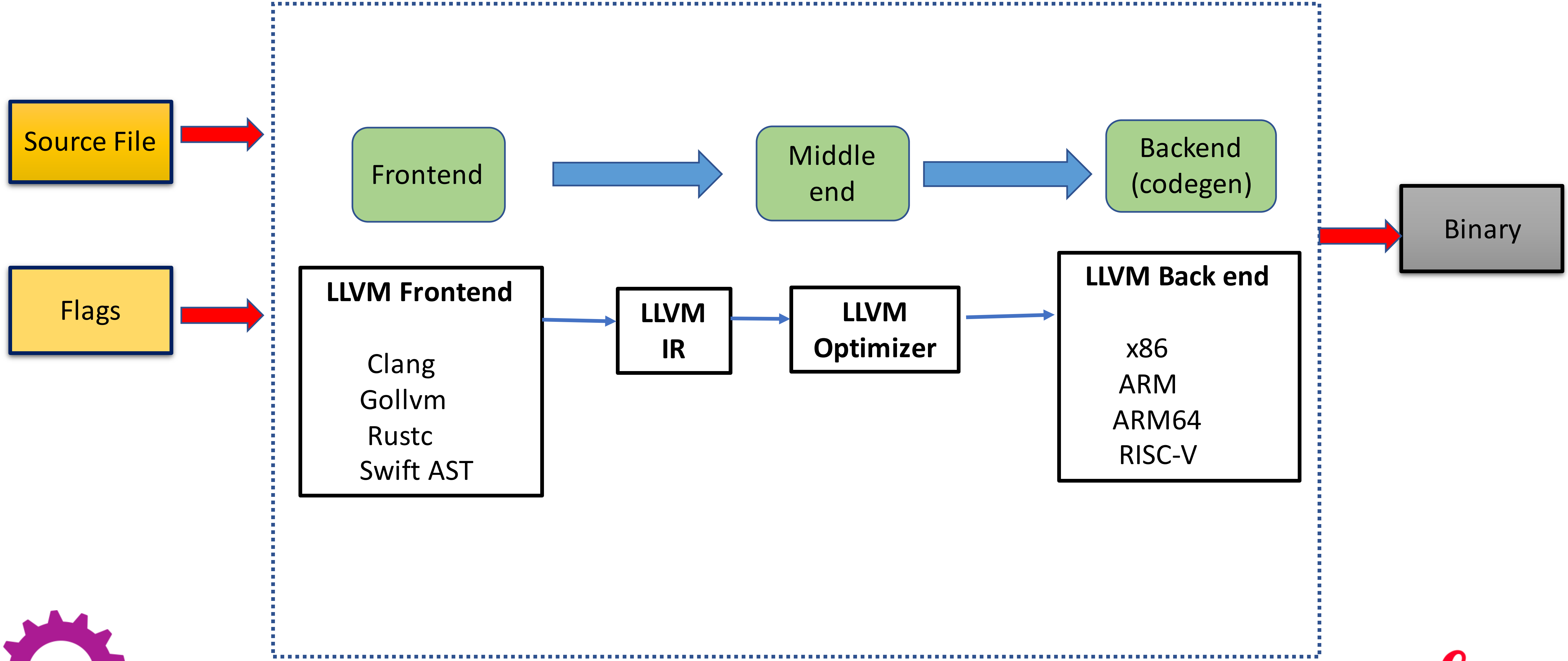
Agenda

- LLVM and Clang
- Compiler-rt Sanitizers
- How to build Compiler-rt Sanitizers
- How Compiler-rt Sanitizers work
- Outlook

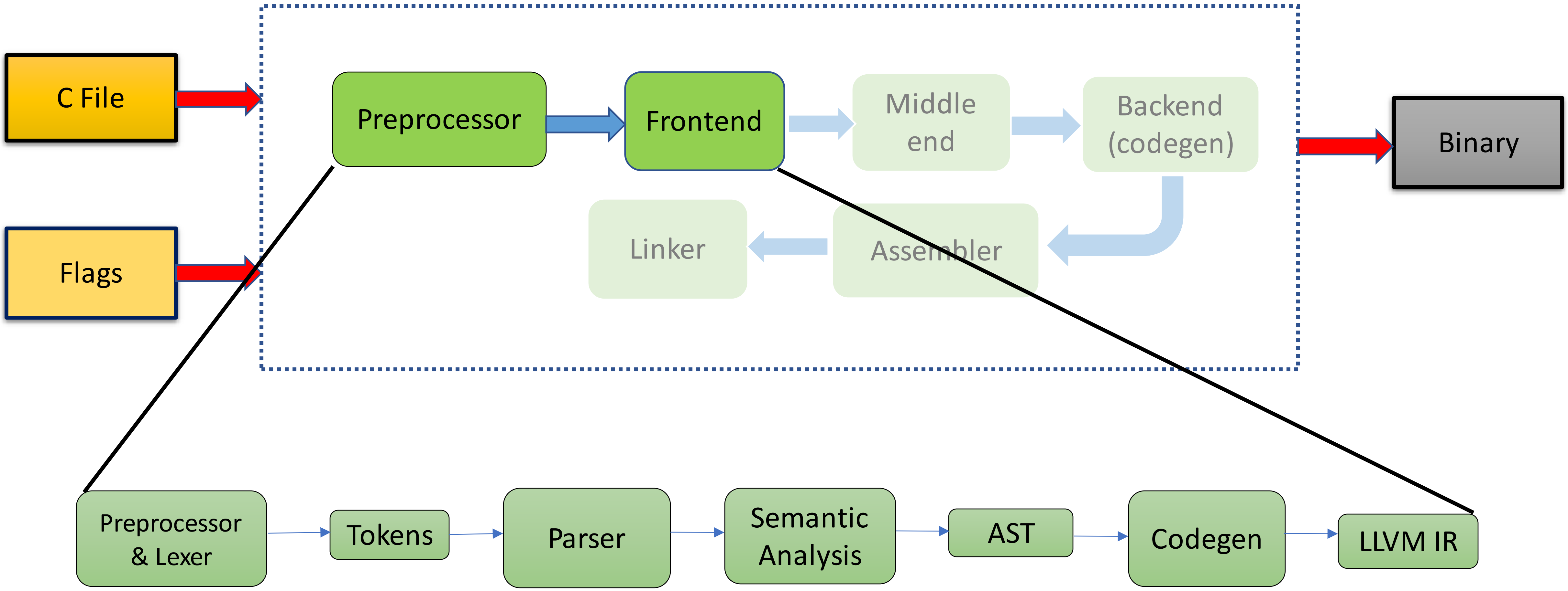
Compiler Flow



LLVM


Source File
Flags
Frontend
Middle end
Backend (codegen)
Binary
LLVM Frontend
Clang
Gollvm
Rustc
Swift AST
LLVM IR
LLVM Optimizer
LLVM Back end
x86
ARM
ARM64
RISC-V

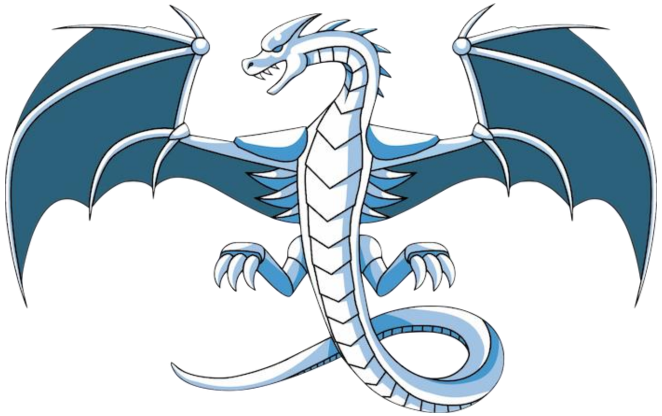
Clang and LLVM



Clang and LLVM

- LLVM Project is a collection of modular and reusable compiler and toolchain technologies. - llvm.org

- Clang is a compiler frontend for C, C++, Objective-C .. in LLVM infrastructure, but
clang (executable)is more than that: **compiler driver**



Leica Geosystems

Compiler-rt Sanitizers

Compiler-rt (Runtimes)

- LLVM equivalent of libgcc.

- It provides target-specific support for low-level functionality that is not supported by the hardware.

- Builtins : provides an implementation of the low-level target-specific hooks required by code generation and other runtime components

- Sanitizers Runtimes: provides instrumentation to catch runtime target behavior like buffer overflow, race conditions, and double-free memory etc.

- Profilers: collect coverage information.

Leica Geosystems

Compiler-rt (Runtimes)

- LLVM equivalent of libgcc.

- It provides target-specific support for low-level functionality that is not supported by the hardware.

- Example:32-bit targets usually lack instructions to support 64-bit division. Let's verify:

```
#include <stdio.h>
#include <stdint.h
#include <stdlib.h>
int main() {
    uint64_t a = 0ULL, b = 0ULL;
    scanf ("%lld %lld", &a, &b);
    printf ("64-bit division is %lld\n", a /
b);
    return EXIT_SUCCESS;
}
```

Compiler-rt

`$clang -S -m32 div_64.c -o div-32.S`

`$clang -S div_64.c -o div-64.S`

Comparing Assembly instruction for both:

```
    movl    -32(%ebp), %esi
    movl    -28(%ebp), %edi
    movl    %esp, %eax
    movl    %edi, 12(%eax)
    movl    %esi, 8(%eax)
    movl    %edx, 4(%eax)
    movl    %ecx, (%eax)
    calll   __udivdi3@PLT
    movl    -36(%ebp), %ebx                 # 4-byte Reload
    movl    %eax, %ecx
    leal    .L.str.1@GOTOFF(%ebx), %eax
    movl    %esp, %eax
    movl    %edx, 8(%eax)
    movl    %ecx, 4(%eax)
    leal    .L.str.1@GOTOFF(%ebx), %ecx
    movl    %ecx, (%eax)
    calll   printf@PLT
    xorl    %eax, %eax
    addl    $44, %esp
    popl    %esi
    popl    %edi
    popl    %ebx
    popl    %ebp
    .cfi_def_cfa %esp, 4
    retl
.Lfunc_end0:
```

```
    callq   __isoc99_scanf@PLT
    movq    -16(%rbp), %rax
    xorl    %ecx, %ecx
    movl    %ecx, %edx
    divq    -24(%rbp)
    movq    %rax, %rsi
    leaq    .L.str.1(%rip), %rdi
    movb    $0, %al
    callq   printf@PLT
    xorl    %eax, %eax
    addq    $32, %rsp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
    .cfi_endproc
                                        # -- End function
    .type   .L.str,@object              # @.str
    .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz  "%ld %ld"
    .size   .L.str, 8

    .type   .L.str.1,@object            # @.str.1
.L.str.1:
```

udivid3: https://github.com/llvm/llvm-project/blob/main/compiler-rt/lib/builtins/i386/udivdi3.S

Compiler-rt Sanitizers

- A sanitizer checks certain runtime properties of the code (**probe**) that's inserted by the compiler. It is used to verify program correctness or check security flaws.

- In LLVM, this kind of instrumentation is provided with the help of compiler-rt as sanitizers

Compiler-rt Sanitizers

- A sanitizer checks certain runtime properties of the code (**probe**) that's inserted by the compiler. It is used to verify program correctness or check security flaws.

- In LLVM, this kind of instrumentation is provided with the help of compiler-rt as sanitizers

- **ASAN**: Address Sanitizer to detect use-after free, buffer-overflow and leaks
- **UBSAN**: Undefined Behavior Sanitizer
- **MSAN**: Memory Sanitizer
- **TSAN**: Thread Sanitizer to detect race conditions and deadlocks

Address Sanitizer: in action

```
int main() {
    char *x = (char*)malloc(10 * sizeof(char*));
    free(x);
    return x[5];
}
```

```
$ clang -rtlib=compiler-rt -fsanitize=address -O1 -fno-omit-frame-pointer -g test.c -o test
$./test
```

```
=================================================================
==175204==ERROR: AddressSanitizer: heap-use-after-free on address 0x3e500ed5 at pc 0x400f6184 bp 0x3fffdcc0 sp 0x3fffdcbc
READ of size 1 at 0x3e500ed5 thread T0

0x3e500ed5 is located 5 bytes inside of 40-byte region [0x3e500ed0,0x3e500ef8)
freed by thread T0 here:
    #0 0x400af3c0 in free /bsp-user-workspaces/shuk/krea-clang/build/tmp/work-shared/llvm-project-source-14.0.3-r0/git/compiler-rt/lib/asan/asan_malloc_linux.cpp:52:3
    #1 0x400f614c in main /home/mamta/kirkstone-sanitizer/test.c:5:6
    #2 0x3f5918ec  (/lib/libc.so.6+0x218ec) (BuildId: 731aa50bb42faaabe5f5ec6aacfc194242150c5f)
    #3 0x3f5919f4  (/lib/libc.so.6+0x219f4) (BuildId: 731aa50bb42faaabe5f5ec6aacfc194242150c5f)

previously allocated by thread T0 here:
    #0 0x400af5f0 in __interceptor_malloc /bsp-user-workspaces/shuk/krea-clang/build/tmp/work-shared/llvm-project-source-14.0.3-r0/git/compiler-rt/lib/asan/asan_malloc_linux.cpp:69:3
    #1 0x400f6144 in main /home/mamta/kirkstone-sanitizer/test.c:4:21
    #2 0x3f5918ec  (/lib/libc.so.6+0x218ec) (BuildId: 731aa50bb42faaabe5f5ec6aacfc194242150c5f)
    #3 0x3f5919f4  (/lib/libc.so.6+0x219f4) (BuildId: 731aa50bb42faaabe5f5ec6aacfc194242150c5f)

SUMMARY: AddressSanitizer: heap-use-after-free /home/mamta/kirkstone-sanitizer/test.c:6:10 in main
```

How to build Compiler-rt Sanitizers?

LLVM with compiler-rt

- Build with LLVM

```
-DLLVM_ENABLE_PROJECTS="clang;compiler-rt"
```

Or

```
-DLLVM_ENABLE_RUNTIMES="compiler-rt"
```

- Separate Build
  Require llvm-config, build llvm first

```
$cmake ../compiler-rt -DLLVM_CONFIG_PATH=</path/to/llvm-config> -G <generator> -B <build_dir>
```

Use any Generator – Ninja or Unix Makefiles

```
$ninja -C <build_dir>
```

Enabling sanitizers with compiler-rt

`With clang build:`

```
$cmake -B build -G Ninja  -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -DCOMPILER_RT_BUILD_SANITIZERS=ON -DLLVM_TARGETS_TO_BUILD=X86 LLVM_OPTIMIZED_TABLEGEN=ON -DCMAKE_BUILD_TYPE=Release llvm/
```

Generated config:

```
-- Builtin supported architectures: x86_64
-- Generated Sanitizer SUPPORTED_TOOLS list on "Linux" is "asan;lsan;msan;tsan;ubsan"
-- sanitizer_common tests on "Linux" will run against "asan;lsan;msan;tsan;ubsan"
-- Supported architectures for crt: x86_64
```

Leica Geosystems

Enabling sanitizers with compiler-rt

With clang build:

$cmake -B build -G Ninja -DLLVM_ENABLE_PROJECTS="clang;compiler-rt" -DCOMPILER_RT_BUILD_SANITIZERS=ON -DLLVM_TARGETS_TO_BUILD=X86 LLVM_OPTIMIZED_TABLEGEN=ON -DCMAKE_BUILD_TYPE=Release llvm/

After build and installation:

```
clang_rt.crtbegin.o          libclang_rt.dfsan.a                  libclang_rt.hwasan.a.syms        libclang_rt.msan.a.syms            libclang_rt.scudo_standalone_cxx.a  libclang_rt.ubsan_standalone.a.syms
clang_rt.crtend.o            libclang_rt.dfsan.a.syms             libclang_rt.hwasan_cxx.a         libclang_rt.msan_cxx.a             libclang_rt.scudo_standalone.so     libclang_rt.ubsan_standalone_cxx.a
libclang_rt.asan.a           libclang_rt.dyndd.so                 libclang_rt.hwasan_cxx.a.syms    libclang_rt.msan_cxx.a.syms        libclang_rt.stats.a                 libclang_rt.ubsan_standalone_cxx.a.syms
libclang_rt.asan.a.syms      libclang_rt.fuzzer.a                 libclang_rt.hwasan-preinit.a     libclang_rt.orc.a                  libclang_rt.stats_client.a          libclang_rt.ubsan_standalone.so
libclang_rt.asan_cxx.a       libclang_rt.fuzzer_interceptors.a    libclang_rt.hwasan.so            libclang_rt.profile.a              libclang_rt.tsan.a                  libclang_rt.xray.a
libclang_rt.asan_cxx.a.syms  libclang_rt.fuzzer_no_main.a         libclang_rt.lsan.a               libclang_rt.safestack.a            libclang_rt.tsan.a.syms             libclang_rt.xray-basic.a
libclang_rt.asan-preinit.a   libclang_rt.gwp_asan.a               libclang_rt.memprof.a            libclang_rt.scudo.a                libclang_rt.tsan_cxx.a              libclang_rt.xray-fdr.a
libclang_rt.asan.so          libclang_rt.hwasan.a                 libclang_rt.memprof.a.syms       libclang_rt.scudo_cxx.a            libclang_rt.tsan_cxx.a.syms         libclang_rt.xray-profiling.a
libclang_rt.asan_static.a    libclang_rt.hwasan_aliases.a         libclang_rt.memprof_cxx.a        libclang_rt.scudo_cxx_minimal.a    libclang_rt.tsan.so
libclang_rt.builtins.a       libclang_rt.hwasan_aliases.a.syms    libclang_rt.memprof_cxx.a.syms   libclang_rt.scudo_minimal.a        libclang_rt.ubsan_minimal.a
libclang_rt.cfi.a            libclang_rt.hwasan_aliases_cxx.a     libclang_rt.memprof-preinit.a    libclang_rt.scudo_minimal.so       libclang_rt.ubsan_minimal.a.syms
libclang_rt.cfi_diag.a       libclang_rt.hwasan_aliases_cxx.a.syms libclang_rt.memprof.so          libclang_rt.scudo.so               libclang_rt.ubsan_minimal.so
libclang_rt.dd.a             libclang_rt.hwasan_aliases.so        libclang_rt.msan.a               libclang_rt.scudo_standalone.a     libclang_rt.ubsan_standalone.a
```

Enabling sanitizers with compiler-rt

With standalone build:

```
$cmake -B build-compiler-rt compiler-rt -DLLVM_CONFIG_PATH=build/bin/llvm-config
 -DCOMPILER_RT_BUILD_SANITIZERS=ON -G Ninja
```

Generated config:

```
Call Stack (most recent call first):
  CMakeLists.txt:71 (load_llvm_config)


-- LLVM_MAIN_SRC_DIR: "/home/mamta/fosdem/llvm-project/llvm"
-- Compiler-RT supported architectures: x86_64
-- Builtin supported architectures: x86_64
-- For x86_64 builtins preferring i386/fp_mode.c to fp_mode.c
-- For x86_64 builtins preferring x86_64/floatdidf.c to floatdidf.c
-- For x86_64 builtins preferring x86_64/floatdisf.c to floatdisf.c
-- For x86_64 builtins preferring x86_64/floatundidf.S to floatundidf.c
-- For x86_64 builtins preferring x86_64/floatundisf.S to floatundisf.c
-- For x86_64 builtins preferring x86_64/floatdixf.c to floatdixf.c
-- For x86_64 builtins preferring x86_64/floatundixf.S to floatundixf.c
-- Supported architectures for crt: x86_64
-- Configuring done
-- Generating done
```

Leica
Geosystems

Cross-compiling compiler-rt sanitizers

$ cmake with options

```
-G Ninja
-DCMAKE_AR=/path/to/llvm-ar
-DCMAKE_ASM_COMPILER_TARGET="arm-linux-gnueabihf"
-DCMAKE_ASM_FLAGS="build-c-flags"
-DCMAKE_C_COMPILER=/path/to/clang
-DCMAKE_C_COMPILER_TARGET="arm-linux-gnueabihf"
-DCMAKE_C_FLAGS="build-c-flags"
-DCMAKE_EXE_LINKER_FLAGS="-fuse-ld=lld"
-DCMAKE_NM=/path/to/llvm-nm
-DCMAKE_RANLIB=/path/to/llvm-ranlib
-DCOMPILER_RT_BUILD_BUILTINS=ON
-DCOMPILER_RT_BUILD_LIBFUZZER=ON
-DCOMPILER_RT_BUILD_MEMPROF=ON
-DCOMPILER_RT_BUILD_PROFILE=ON
-DCOMPILER_RT_BUILD_SANITIZERS=ON
-DCOMPILER_RT_BUILD_XRAY=OFF
-DCOMPILER_RT_DEFAULT_TARGET_ONLY=ON
-DLLVM_CONFIG_PATH=/path/to/llvm-config
```

Making it a bit easier for embedded devices: meta-clang

Leica
Geosystems

Compiler-rt sanitizers in meta-clang

- Add meta-clang layer – lldb, cross-compiler, sanitizer

```
$ bitbake-layers add-layer meta-clang
```

Yoe Distro CI failing

meta-clang (C/C++ frontend and LLVM compiler backend)

This layer provides clang/llvm as alternative to system C/C++ compiler for OpenEmbedded/Yocto Project based distributions. This can cohabit with GNU compiler and can be used for specific recipes or full system compiler.

Getting Started

```
git clone git://github.com/openembedded/openembedded-core.git
cd openembedded-core
git clone git://github.com/openembedded/bitbake.git
git clone git://github.com/kraj/meta-clang.git

$ . ./oe-init-build-env
```

Compiler-rt sanitizers in meta-clang

- To build clang SDK:
  `CLANGSDK=1` in local.conf

- To use clang as default toolchain:
  `TOOLCHAIN = "clang"`

- To use LLVM Runtime:
  `RUNTIME = "llvm"`

- To build compiler-rt and compiler-rt-sanitizer in SDK add in local.conf or in packagegroups:

```
TOOLCHAIN_HOST_TASK:append = " nativesdk-compiler-rt nativesdk-compiler-rt-sanitizers
TOOLCHAIN_TARGET_TASK:append = " compiler-rt-dev compiler-rt-staticdev compiler-rt sanitizers-dev compiler-rt-sanitizers-staticdev"
```

Compiler-rt sanitizers in meta-clang

- To use SDK:

```
$ sh oecore-x86_64-cortexa15t2hf-neon-toolchain-nodistro.0.sh  //Install SDK
$ source environment-setup-cortexa9hf-neon-poky-linux-gnueabi
```

- To cross-compile with clang :

```
$ {CLANGCC} -rtlib=compiler-rt -fsanitize=address  test-sanitizer.c -o test
```

- To test in Qemu :

```
$ qemu-arm  -L <path_to_sysroot> ./test-sanitizer –v
```

How Compiler-rt Sanitizers work?

Address Sanitizer: a deep dive

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)
   buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
   printf("%d ", buffer[i-1]);
  return 0;
}
```

Compile without sanitizer and run:

```
$clang  test-overflow.c -o test
./test 1 2 4 4
Segmentation fault (core dumped)
```

Address Sanitizer: a deep dive

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)
   buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
   printf("%d ", buffer[i-1]);
  return 0;
}
```

Compile and run:

```
$clang -rtlib=compiler-rt -fsanitize=address -O1 -fno-omit-frame-pointer
  test-overflow.c -o test
```

Address Sanitizer: a deep dive

```
=================================================================
==19270==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x3fffdb38 at pc 0x400f6418
bp 0x3fffdb18 sp 0x3fffdb14
WRITE of size 4 at 0x3fffdb38 thread T0
    #0 0x400f6414 in main /home/mamta/fosdem/sdk-test/test-overflow.c:9:17
    #1 0x3f5818ec  (/lib/libc.so.6+0x218ec) (BuildId: d34a05151f021dd285dc5f185d4029a0d135ab64)
    #2 0x3f5819f4  (/lib/libc.so.6+0x219f4) (BuildId: d34a05151f021dd285dc5f185d4029a0d135ab64)

Address 0x3fffdb38 is located in stack of thread T0 at offset 24 in frame
    #0 0x400f62a4 in main /home/mamta/fosdem/sdk-test/test-overflow.c:4

  This frame has 1 object(s):
    [16, 24) 'buffer' (line 6) <== Memory access at offset 24 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism,
swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/mamta/fosdem/sdk-test/test-
overflow.c:9:17 in main
```

Geosystems

Address Sanitizer: a deep dive

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)
   buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
   printf("%d ", buffer[i-1]);
  return 0;
}
```

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)

    if (i>2){
        notifyerror();
    }

  buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
  printf("%d ", buffer[i-1]);
 return 0;
}
```

After adding the sanitizer instrumentation

Address Sanitizer: a deep dive

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)
   buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
   printf("%d ", buffer[i-1]);
  return 0;
}
```

```
int main(int argc, char **argv) {
int buffer[2];
for (int i = 1; i < argc; ++i)

    if (i>2) {
          notifyerror();
    }

  buffer[i-1] = atoi(argv[i]);
for (int i = 1; i < argc; ++i)
  printf("%d ", buffer[i-1]);
 return 0;
}
```

```
if (IsPoisoned(buffer)) {
  ReportError(buffer, kAccessSize, kIsWrite);
}
```

Memory that shouldn’t be accessed is **poisoned**

Address Sanitizer: a deep dive

```
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/mamta/fosdem/sdk-test/test-overflow.c:9:17 in main
Shadow bytes around the buggy address:
  0x27fffb10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x27fffb60: 00 00 00 00 f1 f1 00[f3]f3 f3 00 00 00 00 00 00
  0x27fffb70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffba0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffbb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
```

Shadow Memory and
Application Memory

Address Sanitizer: a deep dive

```
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/mamta/fosdem/sdk-test/test-overflow.c:9:17 in main
Shadow bytes around the buggy address:
  0x27fffb10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x27fffb60: 00 00 00 00 f1 f1 00[f3]f3 f3 00 00 00 00 00 00
  0x27fffb70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffb90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffba0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x27fffbb0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
```

Shadow Memory and Application Memory

Outlook

Outlook

- Great tool to find bugs in runtime for complex applications
- By using sanitizers, we can improve development quality with ease and with high precision
- Increases code size but still faster than Valgrind
- Still not all architectures are supported uniformly

Questions ?

Write to me or connect