Debugging With LLVM
A quick introduction to LLDB and LLVM sanitizers

Graham Hunter, Andrzej Warzyński

Arm

February 2020
Our Background

- Compiler engineers at Arm
  - Arm Compiler For Linux
  - Downstream and upstream LLVM
  - Based in Manchester, UK

- Scalable Vector Extension (SVE) for AArch64

- OpenMP Committee Member (Graham)

- LLDB developer in previous life (Andrzejj)
LLDB - Architecture

LLDB offers multiple options:

- **user drivers**: command line, lldb-mi, Python
- **debug API**: ptrace/simulator/runtime/actual drivers
GDB Remote Serial Protocol

- Simple, ASCII message based protocol
- Designed for debugging remote targets
- Extended for LLDB, see lldb-gdb-remote.txt

**GDB RSP packet structure:**

```
$ ... # h h
```

**Debugging:**

(lldb) log enable gdb-remote packets
(lldb) log list
LLDB command structure

- LLDB command syntax is fairly structured:

  ```
  (lldb) <noun> <verb> [-options [option-value]] [argument [argument...]]
  ```

- For example:

  ```
  (lldb) breakpoint set --file foo.c --line 12
  (lldb) process launch --stop-at-entry -- -program_arg value
  ```

- When in doubt:

  ```
  (lldb) apropos <keyword>
  ```
## GDB to LLDB command map

<table>
<thead>
<tr>
<th>GDB</th>
<th>LLDB</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>% gdb -args a.out 1 2 3</code></td>
<td><code>% ll pdb -a.out 1 2 3</code></td>
</tr>
<tr>
<td><code>(gdb) run</code></td>
<td><code>(lldb) process launch - &lt;args&gt;</code></td>
</tr>
<tr>
<td><code>(gdb) r</code></td>
<td><code>(lldb) run &lt;args&gt;</code></td>
</tr>
<tr>
<td><code>(gdb) step</code></td>
<td><code>(lldb) thread step-in</code></td>
</tr>
<tr>
<td><code>(gdb) s</code></td>
<td><code>(lldb) step</code></td>
</tr>
<tr>
<td><code>(gdb) next</code></td>
<td><code>(lldb) thread step-over</code></td>
</tr>
<tr>
<td><code>(gdb) n</code></td>
<td><code>(lldb) next</code></td>
</tr>
<tr>
<td><code>(gdb) break main</code></td>
<td><code>(lldb) breakpoint set -name main</code></td>
</tr>
<tr>
<td></td>
<td><code>(lldb) br s -n main</code></td>
</tr>
<tr>
<td></td>
<td><code>(lldb) b main</code></td>
</tr>
</tbody>
</table>
### GDB to LLDB command map

<table>
<thead>
<tr>
<th>GDB</th>
<th>LLDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gdb) break test.c:12</td>
<td>(lldb) breakpoint set -file test.c -line 12</td>
</tr>
<tr>
<td></td>
<td>(lldb) br s -f test.c -l 12</td>
</tr>
<tr>
<td></td>
<td>(lldb) b test.c:12</td>
</tr>
<tr>
<td>(gdb) info break</td>
<td>(lldb) breakpoint list</td>
</tr>
<tr>
<td></td>
<td>(lldb) br l</td>
</tr>
<tr>
<td>(gdb) set env DEBUG 1</td>
<td>(lldb) settings set target.env-vargs DEBUG=1</td>
</tr>
<tr>
<td></td>
<td>(lldb) set se target.env-vargs DEBUG=1</td>
</tr>
<tr>
<td></td>
<td>(lldb) env DEBUG=1</td>
</tr>
<tr>
<td>(gdb) show args</td>
<td>(lldb) settings show target.run-args</td>
</tr>
</tbody>
</table>

- More at: [https://lldb.llvm.org/use/map.html](https://lldb.llvm.org/use/map.html)
Beyond basic usage

- **Evaluating expressions:**
  
  ```
  (lldb) expr (int) printf("Print nine: %d.", 4 + 5)
  ```

- **Python interpreter:**
  
  ```
  (lldb) script
  >>> import os
  >>> print("I am running on pid ".format(os.getpid()))
  ```

- **Custom commands:**
  
  ```
  (lldb) command script add -f my_commands.printworld hello
  ```
LLDB links

- LLDB Tutorial: https://lldb.llvm.org/use/tutorial.html


- llvm-tutor: https://github.com/banach-space/llvm-tutor/
Part 2

LLVM Sanitizers
Binary Instrumentation to aid Debugging
Binary Instrumentation to aid Debugging

clang -g -O1 -fsanitize=address my_prog.c -o my_prog

• Several sanitizers available to target different possible bugs, e.g. address (ASAN), thread (TSAN), memory (MSAN)
• Wraps various operations in your code (e.g. memory traffic)
Binary Instrumentation to aid Debugging

```bash
clang -g -O1 -fsanitize=address my_prog.c -o my_prog
```

- Several sanitizers available to target different possible bugs, e.g. address (ASAN), thread (TSAN), memory (MSAN)
- Wraps various operations in your code (e.g. memory traffic)
- Tunable behavior on encountering a problem

<table>
<thead>
<tr>
<th><code>-fsanitize=</code></th>
<th>Print verbose error, <strong>continue</strong> execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-fno-sanitize-recover=</code></td>
<td>Print verbose error, <strong>terminate</strong> program</td>
</tr>
<tr>
<td><code>-fsanitize-trap=</code></td>
<td>Execute a <strong>trap</strong> instruction (only for ubsan)</td>
</tr>
</tbody>
</table>
Binary Instrumentation to aid Debugging

```bash
clang -g -O1 -fsanitize=address my_prog.c -o my_prog
```

- Several sanitizers available to target different possible bugs, e.g. address (ASAN), thread (TSAN), memory (MSAN)
- Wraps various operations in your code (e.g. memory traffic)
- Tunable behavior on encountering a problem

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-fsanitize=</code></td>
<td>Print verbose error, <strong>continue</strong> execution</td>
</tr>
<tr>
<td><code>-fno-sanitize-recover=</code></td>
<td>Print verbose error, <strong>terminate</strong> program</td>
</tr>
<tr>
<td><code>-fsanitize-trap=</code></td>
<td>Execute a <strong>trap</strong> instruction (only for ubsan)</td>
</tr>
</tbody>
</table>

- Can be combined

  ```bash
  -fsanitize=signed-integer-overflow -fno-sanitize-recover=address
  ```
Binary Instrumentation to aid Debugging

```
clang -g -O1 -fsanitize=address my_prog.c -o my_prog
```

- Several sanitizers available to target different possible bugs, e.g. address (ASAN), thread (TSAN), memory (MSAN)
- Wraps various operations in your code (e.g. memory traffic)
- Tunable behavior on encountering a problem

| `-fsanitize=` | Print verbose error, **continue** execution |
| `--fno-sanitize-recover=` | Print verbose error, **terminate** program |
| `--fsanitize-trap=` | Execute a **trap** instruction (only for ubsan) |

- Can be combined

```
-fsanitize=signed-integer-overflow -fno-sanitize-recover=address
```

- ASAN, MSAN, and TSAN are **mutually exclusive!**
Address Sanitizer (ASAN)

**main.c**

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define ARRAY_ELTS (10)
#define ARRAY_SIZE (sizeof(int) * ARRAY_ELTS)

extern int my_loop(int*, int);

int main(int argc, char **argv) {
    int *array = (int*)malloc(ARRAY_SIZE);
    memset(array, 0, ARRAY_SIZE);
    int result = my_loop(array, ARRAY_SIZE);

    printf("Result was: %d\n", result);
    return 0;
}
```

**loop.c**

```c
int my_loop(int *array, int num_elems) {
    int result = 0;
    for (int i = 0; i < num_elems; i++) {
        // Some expensive calculation not shown
        // here
        result += array[i];
    }
    return result;
}
```
Address Sanitizer (ASAN)

- Detects out-of-bounds accesses, use-after-free/scope, double free
- Option to detect leaks (on by default on Linux)

```
ASAN_OPTIONS=detect_leaks=1 ./my_instrumented_binary
```

- Option to detect initialization order problem (Linux only)

```
ASAN_OPTIONS=check_initialization_order=1 ./my_instrumented_binary
```
Undefined Behavior Sanitizer

- Catches several cases of UB in C and C++
- Can also catch similar cases that are not technically UB but may still be undesirable
Undefined Behavior Sanitizer

Unsigned integer wrapping

```c
#include <stdio.h>
#include <stdint.h>

unsigned getSizeOfA() {
    return 8;
}

unsigned getSizeOfB() {
    return 32;
}

int main(int argc, char **argv) {
    int64_t Offset = 0;

    Offset = (getSizeOfA() - getSizeOfB()) / 8 - Offset;
    printf("Offset %lld, Offset in Bits: %lld\n", Offset, Offset * 8);
    return 0;
}
```
#include <pthread.h>
#include <stdio.h>

int *item = NULL;
int someval = 5;
int ready = 0;

void *thread1(void *x) {
  item = &someval;
  ready = 1;
  return NULL;
}

void *thread2(void *x) {
  if (!ready)
    return NULL;
  int val = *item;
  // Process item here.
  return NULL;
}

int main() {
  int val = 0;
  pthread_t t0, t1;
  pthread_create(&t0, NULL, thread1, NULL);
  pthread_create(&t1, NULL, thread2, NULL);
  pthread_join(t0, NULL);
  pthread_join(t1, NULL);
  return 0;
}
Thread Sanitizer (TSAN)

- Detects data races, including on mutexes themselves (lock in one thread before init in another)
- Catches destruction of a mutex while still locked
- Catches signal handlers overwriting errno
- Can annotate the source to indicate correctness (ANNOTATE_HAPPENS BEFORE, etc)
- Can report more history if required (2 is the default, 7 the max)

```
TSAN_OPTIONS="history_size=4" ./my_instrumented_binary
```
int main(int argc, char **argv) {
    int opt = atoi(argv[1]);
    int foo;

    switch (opt) {
    case 0:
        foo = 3;
        break;
    case 1:
        foo = 8;
        break;
    }

    printf("Foo is: %d\n", foo);
    return 0;
}
Memory Sanitizer (MSAN)

- Catches reads of uninitialized memory
- Only supports Linux/FreeBSD/NetBSD at present
- Can track origins of memory

```
-fsanitize=memory -fsanitize-memory-track-origins=2
```
More Precise Configuration

- May be too much overhead to instrument entire program, want to exclude hot code

- Can suppress in the source

```c
__attribute__((no_sanitize("address")))
```

- May need a more centralized option
Sanitizer Special Case List

List of exclusions provided at compile time

```shell
clang -fsanitize=address -fsanitize-blacklist=exclusions.txt ...
```

```plaintext
#comments
#suppress for any sanitizer by default
src:/path/to/myfile.c
fun:func1
#cpp names mangled
#can suppress for specific sanitizer only with [sections]
src:/path/to/myotherfile.cpp
[address]
fun:_Z9OtherFuncv
#shell wildcard ‘*’ allowed for file and function name matching exclusions.txt
```
More info

- Haven’t covered all of them
  - pointer-compare, pointer-subtract – detect UB on pointer comparisons for different objects
  - control-flow integrity (cfi) – catches corruption of branch addresses
  - dfsan – manual annotation of data flow
  - More being written – TySan under review for catching strict aliasing problems

- [https://clang.llvm.org/docs/index.html](https://clang.llvm.org/docs/index.html)
  - Links to documentation for several sanitizers and other built-in analysis and instrumentation tools

- [https://github.com/google/sanitizers/wiki](https://github.com/google/sanitizers/wiki)
  - Google’s sanitizer wiki; old, but still contains some useful info

- Has been used in public CI instances (e.g. Travis)
Final thoughts

- LLDB is a very mature debugger
  - It is very likely already available on your platform

- LLVM’s sanitizers are very powerful, yet straightforward to use
  - No extra tools required - just add `-fsanitize=` when building

- You can use sanitisers from inside LLDB:

  ```
  (lldb) memory history <address>
  ```

  andrzej.warzynski@arm.com, graham.hunter@arm.com