llvm-crash-analyzer

A Tool for the Program Analysis of Corefiles

Djordje Todorovic[1,2], Bharathi Seshadri[1], Ananth Sowda[1], Nikola Tesic[1,2], Ivan Baev[1,3]

CGO Workshop, 3rd April 2022

[1] Cisco Systems [2] Syrmia [3] SiFive

Topics

- Motivation
- Design Goals / Architecture
- Modules
- Demo
- Conclusion & Future Work

Crash Debugging/Triaging Workflow

- Live debugging using debugger
- Offline debugging using corefile
  - Get list of function/file in crash backtrace
  - Route bugs to the right component owner
  - Look for unusual argument values or control flow

```
(gdb) bt
#0  input_cfg (fn=<optimized out>, data_in=0x2cf0700, ib=0x7fffffffd3c0) at ../../gcc/gcc/lto-streamer-in.cc:1019
#1  input_function (node=<optimized out>, ib_cfg=0x7fffffffd3c0, ib=0x7fffffffd3a0, data_in=0x2cf0700, fn_decl=<optimized out>) at ../../gcc/gcc/lto-streamer-in.cc:1392
#2  lto_read_body_or_constructor (file_data=<optimized out>, node=<optimized out>, data=<optimized out>, section_type=LTO_section_function_body) at ../../gcc/gcc/lto-streamer-in.cc:1621
#3  0x000000000098ba99 in cgraph_node::get_untransformed_body (this=this@entry=0x7ffff67ce000) at ../../gcc/gcc/cgraph.cc:4002
#4  0x0000000001be0075 in get_whitelisted_nodes () at ../../gcc/gcc/ipa-type-escape-analysis.c:282
#5  0x0000000001be2408 in lto_dead_field_elimination () at ../../gcc/gcc/ipa-type-escape-analysis.c:357
#6  0x0000000001be2d39 in lto_dfe_execute () at ../../gcc/gcc/ipa-type-escape-analysis.c:247
#7  (anonymous namespace)::pass_ipa_type_escape_analysis::execute (this=<optimized out>) at ../../gcc/gcc/ipa-type-escape-analysis.c:238
#8  0x0000000000d3dd6b in execute_one_pass (pass=0x2c4d800) at ../../gcc/gcc/passes.cc:2637
#9  0x0000000000d3f0d7 in execute_ipa_pass_list (pass=0x2c4d800) at ../../gcc/gcc/passes.cc:3077
#10 0x00000000000997d4e in symbol_table::compile (this=0x7ffff69a1000) at ../../gcc/gcc/context.h:48
#11 0x00000000008f115a in lto_main () at ../../gcc/gcc/lto/lto.cc:653
#12 0x0000000000e1ef3e in compile_file () at ../../gcc/gcc/toplev.cc:452
#13 0x00000000008bee59 in do_compile (no_backend=false) at ../../gcc/gcc/toplev.cc:2158
#14 toplev::main (this=this@entry=0x7fffffffde0e, argc=<optimized out>, argc@entry=3, argv=<optimized out>, argv@entry=0x7fffffffdf18) at ../../gcc/gcc/toplev.cc:2310
#15 0x00000000008c05ab in main (argc=3, argv=0x7fffffffdf18) at ../../gcc/gcc/main.cc:39
```

How to get more insights into the root cause?

Background: Retracer

- Tool developed by Microsoft for Windows (proprietary)
- Triage software crashes using memory dumps
- Help QA engineers when filing bugs
- Finding root cause of the crash and grouping the crashes by module of the blame function
- Backward analysis

Objectives

- Assist with fault localization
  - Symptom: Segfault due to an invalid pointer address
  - Goal: Identify function responsible for a bad value
- Reuse existing code
  - Leverage LLVM libraries and tools as building blocks

Expectations

Effective

- Deterministic errors such as SEGV, BUS, ABRT
- Blame code is in the crashing thread and in the backtrace (BT)
- Blame code is NOT in BT, but called by one of the functions in BT

Not addressed

- Blame code belongs to some other thread
- Memory corruption issues
- Memory allocation errors
- Stack overflow errors
- Race condition, timing issues

Architecture of llvm-crash-analyzer

Core Reader - extracts functions, registers, memory state from the backtrace of the crashing thread.

Decompiler - for all functions in backtrace, produces Machine IR, register-state, and global state.

Analyzer - performs forward and backward compiler analyses (with help of the Concrete Reverse Execution) at Machine IR level. Reports “blame” function on success.

Example

```
void f() {
  T p;
  p.fn = NULL; // blame point
  g(&p);
}
int main() {
 f();
 return 0;
}
```

```
void h(int *r) {
  *r = 3; // crash
}

void g (T*q) {
  int *t = q->fn;
  h(t);
}
```

Backtrace:

#0 h (r=0x0) at test.c:7
#1 g (q=0x7ffd71b147e8) at test.c:12
#2 f () at test.c:20
#3 main () at test.c:24

```
[ntesic@sjc-ads-5118 basic-test]$ ../../build_dev/bin/llvm-crash-analyzer --core-file=core.basic.1142843 basic
Crash Analyzer -- crash analyzer utility

Loading core-file core.basic.1142843
core-file processed.

Decompiling...
Decompiling h
Decompiling g
Decompiling f
Decompiling main
Decompiled.

Analyzing...

Blame Function is f
From File /nobackup/ntesic/PAC/CGO/basic-test/basic-test.c
```

Corefile Reader

Corefile Reader

- LLDB library
- Read the backtrace
- map/remember register values of each frame at the moment of the crash
  - This info is stored into specific MachineFunction attribute of the decompiled LLVM MIR Function
- Debugging:
  - --debug-only=llvm-crash-analyzer-corefile

Decompilation

Compilation

- LLVM MIR – Target dependent code representation
  - Machine Function attributes
  - Registers/Register state flags
  - CFG
  - Data Flow Analysis

```
# Machine code for function foo: NoPHIs, TracksLiveness, NoVRegs, TiedOpsRewritten
bb.0.entry:
  frame-setup PUSH64r undef $rax, implicit-def $rsp, implicit $rsp
  CFI_INSTRUCTION def_cfa_offset 16
  dead $eax = XOR32rr undef $eax(tied-def 0), undef $eax, implicit-def dead $eflags, implicit-def $al
  CALL64pcrel32 @baa, <regmask $bh $bl $bp $bph $bpl $bx $ebp $ebx $hbp $hbx $rbp $rbx $r12 $r13
$r14 $r15 $r12b $r13b $r14b $r15b $r12bh $r13bh $r14bh $r15bh $r12d $r13d $r14d $r15d $r12w $r13w
$r14w $r15w $r12wh and 3 more...>, implicit $rsp, implicit $ssp, implicit $al, implicit-def $rsp, implicit-def
$ssp, implicit-def $eax
  renamable $ecx = XOR32rr undef $ecx(tied-def 0), undef $ecx, implicit-def dead $eflags
  TEST32rr killed renamable $eax, renamable $eax, implicit-def $efl
  renamable $cl = SETCCr 5, implicit killed $eflags, implicit killed $ecx, implicit-def $ecx
  $eax = MOV32rr killed $ecx
  $rcx = frame-destroy POP64r implicit-def $rsp, implicit $rsp
  CFI_INSTRUCTION def_cfa_offset 8
  RETQ $eax
# End machine code for function foo.
```

Decompilation - DisAssembly

- Disassembly
  - Corefile (contains info about .text segment of funcs from backtrace)
  - libLLDB Library
  - Multiple architecture support
    - For now, we are testing x86_64 architecture only
- LLDB debugger also uses the libLLDB library

```
(lldb) bt
* thread #1, name = 'clang-test00', stop reason = signal SIGSEGV
  * frame #0: 0x000000000040111c clang-test00`h(r=0x0000000000000000) at test.c:7:6
    frame #1: 0x0000000000401150 clang-test00`g(q=0x00007ffe7822d248) at test.c:12:3
    frame #2: 0x0000000000401181 clang-test00`f at test.c:20:3
    frame #3: 0x00000000004011a4 clang-test00`main at test.c:24:2
    frame #4: 0x00007f530f84b545 libc.so.6`__libc_start_main + 245
    frame #5: 0x000000000040104e clang-test00`_start + 46
(lldb) disassemble
clang-test00`h:
    0x401110 <+0>:  pushq  %rbp
    0x401111 <+1>:  movq   %rsp, %rbp
    0x401114 <+4>:  movq   %rdi, -0x8(%rbp)
    0x401118 <+8>:  movq   -0x8(%rbp), %rax
->  0x40111c <+12>: movl   $0x0, (%rax)
    0x401122 <+18>: popq   %rbp
    0x401123 <+19>: retq
```

DeCompilation – MIR Level

- Motivation: Using existing LLVM infrastructure
  - MCInst -> MachineInstr (MachineBasicBlock, MachineFunction)
- MIR Module
  - Create IR Module (necessary for creation of Machine IR Module)
  - Create IR Dummy function for each function from backtrace
  - Create Machine IR Module with real Machine Functions (by creating debug info to report real line:column – by using DIBuilder and DI metadata – debug sections needed in the executable)
  - CFG
  - Reconstruct register state flags (we created LLVM Pass - FixRegStateFlags)
  - Create regInfo new Machine Function attribute with register values (read from corefile), so it used during analysis; crash-start MI attribute

DeCompilation to the MIR level

```
$ cat test.mir
--- |
  ; ModuleID = 'clang-test00'
  source_filename = "clang-test00"
  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

  ; Materializable
  define void @f() !dbg !2 {
  entry:
    unreachable
  }
  ...
  !llvm.dbg.cu = !{!0}

  !0 = distinct !DICompileUnit(language: DW_LANG_C, file: !1, producer: "crash-blamer", isOptimized: true, runtimeVersion: 0, emissionKind: FullDebug)
  !1 = !DIFile(filename: "/home/djtodorovic/projects/cpp_examples/crashes/cisco-standard-case/test.c", directory: "/")
  !2 = distinct !DISubprogram(name: "f", linkageName: "f", scope: null, file: !1, line: 1, type: !3, scopeLine: 1, spFlags: DISPFlagDefinition | DISPFlagOptimized, unit: !0, retainedNodes: !4)
  ...
...
```

```
---
name:            h
alignment:       16
regInfo:         { GPRegs:
  - { reg: rax, value: '0x0000000000000000' }
  - { reg: rbx, value: '0x0000000000000000' }
  - { reg: rcx, value: '0x00000000004011b0' }
  - { reg: rdx, value: '0x00007ffe7822d368' }
  - { reg: rdi, value: '0x0000000000000000' }
  - { reg: rsi, value: '0x00007ffe7822d358' }
  - { reg: rbp, value: '0x00007ffe7822d210' }
  - { reg: rsp, value: '0x00007ffe7822d210' }
  - { reg: r8, value: '0x00007f530fbf1e80' }
  - { reg: r9, value: '0x0000000000000000' }
  - { reg: r10, value: '0x000000000000000d' }
  - { reg: r11, value: '0x0000000000000006' }
...
CrashOrder: 1
body:             |
 bb.0:
  liveins: $rbp, $rdi

  PUSH64r $rbp, implicit-def $rsp, implicit $rsp, debug-location !DILocation(line: 6, scope: !6)
  $rbp = MOV64rr $rsp, debug-location !DILocation(line: 6, scope: !6)
  MOV64mr $rbp, 1, $noreg, -8, $noreg, $rdi, debug-location !DILocation(line: 6, scope: !6)
  $rax = MOV64rm $rbp, 1, $noreg, -8, $noreg, debug-location !DILocation(line: 7, column: 4, scope: !6)
  crash-start MOV32mi $rax, 1, $noreg, 0, $noreg, 0, debug-location !DILocation(line: 7, column: 6, scope: !6)
  $rbp = POP64r implicit-def $rsp, implicit $rsp, debug-location !DILocation(line: 8, column: 1, scope: !6)
  RETQ debug-location !DILocation(line: 8, column: 1, scope: !6)
...
```

Decompilation – Debug info

- Debug info
  - DIModule
    - Onto each IR Function attach DISubprogram with info found in .debug_info (e.g. DW_TAG_subprogram)
  - DILocation -- line;column
  - Inlining
    - For each inlined function create a dummy MF with only NOOP instruction by creating a DISubprogram describing the function (so it can be used when creating DILocation and during reporting blame function/line)
      - **When reporting blame function, identify the blame MI, check the debug info attachment onto MI, rather than checking parent (MF) of the MI**

Analysis

Analysis

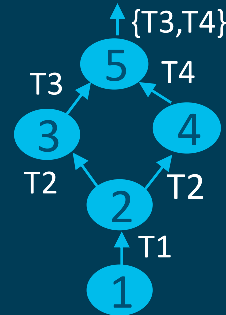
- Goal: Invalid pointer/Address, where did it originate?
- Analysis done on LLVM Machine Intermediate Representation (MIR)
- Backward Analysis: Taint Analysis, Concrete Reverse Execution
- Forward Analysis: Register Equivalence

Backward Taint Analysis

- Post order traversal
- Start Taint (bt #0)
  - Identify crash operands at crash-start
  - Taint Crash Operand
    - Memory Operand only ({base, index})
  - Taint Concrete Memory Address (if known)
    - Using dedicated MIR Function attribute `regInfo`

Taint Propagation

- For each Machine Instruction
  - Examine Source and Destination Operands
  - If Tainted Operand is a Destination
    - Add Source operand to taint list
    - Remove Destination operand
- Merge Taint at the entry of each MBB
- Function calls
  - Propagate taint through called function if
    1. Return value is tainted
    2. Global variable is tainted
  - Identifying if a function modifies a taint operand – not implemented



Backward Taint Analysis : Taint Termination Conditions

- TaintList is empty
- Reached end of backtrace
- If a store instruction has both
    1. Tainted destination operand
    2. Source operand is a constant

Backward Analysis: Concrete Reverse Execution (CRE)

- Concrete Memory Addresses and Concrete reverse execution
  - Motivation: regX + offsetN in frame #1 **not (always) equal** regX + offsetN in frame #2

Concrete Reverse Execution

function: bar
registers:
  rax: ~~0xA5~~ **0xA4**
  rdx: 0xAD
  ...
code:
  ...
→ add $rax, 1
  ...

f^{-1}(add)

Forward Analysis : Register Equivalence (RE)

- Forward Analysis is run before the Backward Analysis
- To help compute concrete memory addresses in more cases
- For each instruction create a list of equivalences between registers

```
…
eax = edx
…
xor eax, eax
…
```

- If the concrete memory address depends on $eax we could use $edx instead by having the RegisterEq table for each instruction

Register Equivalence

- --debug-only=register-eq
- $ llvm-crash-analyzer --core-file=core.a.out.1715642 a.out --debug-only=register-eq

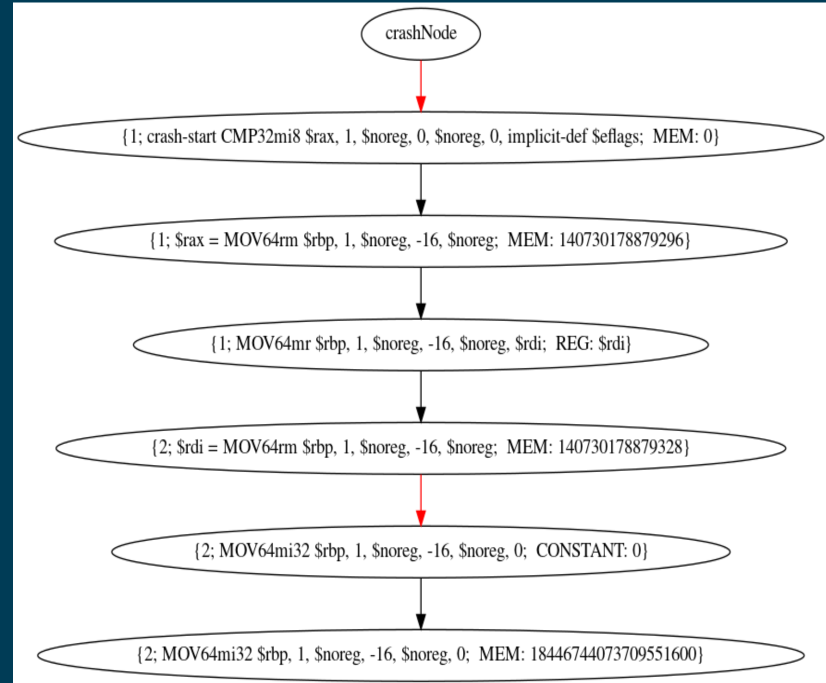
```
*** Register Equivalence Analysis (main)***

** join for bb.0

Reg Eq Table:

$rbp : { $rbp }

Reg Eq Table after: PUSH64r $rbp, implicit-def $rsp, implicit $rsp, debug-location !DILocation(line: 8, scope: !5); test.c:8

$rsp : { $rsp }

Reg Eq Table after: $rbp = MOV64rr $rsp, debug-location !DILocation(line: 8, scope: !5); test.c:8

$rbp : { $rbp $rsp }

$rsp : { $rbp $rsp }

…
```

Backward Taint Graph

- Directed Data flow graph
- Traces the flow of Tainted Operand
- Data flow Node
  - {Frame level, MI, Tainted Operand}
- Root/Crash Node : {1, Crash Instruction, Crash Operand}
- Edges
  - Assignment Edge : Dest Op -> Source Op
  - Dereference Edge : Critical value -> Loc

Finding the Blame Function

- $ llvm-crash-analyzer --core-file=core a.out --print-dfg-as-dot=dfg.dot
- Taint DFG traversal
  - Begin at Crash Node
  - Follow deref edge to identify bad value
  - Follow assignment edge
  - Find deepest node with deref edge



Summary of Implementation & Evaluation

- Work In Progress
- Implemented required building blocks
  - Taint Analysis, CRE, RE, Taint Flow Graph
- Works on several crafted test cases
  - Evaluation with real cases in progress
- Open sourced : https://github.com/cisco-open/llvm-crash-analyzer

llvm-crash-analyzer Repo

- Main Driver (llvm-crash-analyzer.cpp)
- Corefile extraction (lib/Corefile/Corefile.cpp)
- Decompilation (lib/Decompiler/Decompiler.cpp)
- Taint Analysis (lib/Analysis/TaintAnalysis.cpp)
- Register Equivalence (lib/Analysis/RegisterEquivalence.cpp)
- Concrete Reverse Execution (lib/Analysis/ConcreteReverseExec.cpp)
- Taint Flow Graph (lib/Analysis/TaintDataFlowGraph.cpp)

Demo

Conclusion

- Developed llvm-crash-analyzer – a new program analysis tool
  - Attempt to identify function responsible for the crash
  - Can help reduce RCA time
  - Current support for x86_64
- More work needed to achieve production quality
- Collaborators welcome!

- Q & A