Dynamic code analysis tools

Stewart Martin-Haugh (STFC RAL)

Berkeley Software Technical Interchange meeting

Science & Technology Facilities Council

Overview

- Introduction
- Sanitizer tools
- AddressSanitizer
- UndefinedBehaviorSanitizer
- Thread sanitizer
- Coverage
- Thoughts/discussion points

Introduction

- Dynamic analysis: any kind of testing that involves running your code and analysing in more detail than “my code doesn’t crash”
- Within ATLAS our workhorses are
  - FPEAuditor: checks for floating point exception in CPU register, generates WARNING or stack trace
  - Valgrind: run Athena etc (takes several hours), read big log file
  - CPU and memory profiling tools (not discussed today)
- FPEAuditor is so well-integrated it’s easy to forget that it’s an extra tool - has found countless bugs
- Valgrind is unwieldy, mainly because of Athena CPU/memory requirements, but still very useful
- Desire to spend less time running valgrind led me to start investigating other tools (jointly with Rolf Seuster, Scott Snyder and others)

Sanitizers

- Suite of Google open-source tools with names ending in Sanitizer
- Dynamic binary instrumentation to search for various types of error
- Several specialised tools instead of one big suite
- Aim for zero false-positive rate
- Enable via compilation options in gcc and clang
  - **-fsanitize=undefined**, **-fsanitize=address** etc
- Can run in opt builds (this is encouraged by sanitizer authors)
- Very successful in open-source projects

AddressSanitizer demo

- setup gcc >= 4.9
- cat boom.cpp

```
int main(int argc, char **argv) {
  int *array = new int[100];
  int res = array[argc + 100];
  delete [] array;
  return res;
}
```

- g++ -fsanitize=address boom.cpp -g

AddressSanitizer output 1

- Message beginning with AddressSanitizer, then stack trace

```
=================================================================
==7552==ERROR: AddressSanitizer: heap-buffer-overflow on address 0
    x61400000ffd4 at pc 0x400825 bp 0x7fff248f7d80 sp 0x7fff248f7d78
READ of size 4 at 0x61400000ffd4 thread T0
    #0 0x400824 in main /tmp/smh/boom.cpp:3
    #1 0x7f4181b04d5c in __libc_start_main (/lib64/libc.so.6+0x1ed5c)
    #2 0x400678 (/tmp/smh/a.out+0x400678)

0x61400000ffd4 is located 4 bytes to the right of 400-byte region [0
    x61400000fe40,0x61400000ffd0)
allocated by thread T0 here:
    #0 0x7f418267e00f in operator new[](unsigned long) (/afs/cern.ch/
        atlas/offline/external/LCG_71san/gcc/4.9.1/x86_64-slc6/lib64/
        libasan.so.1+0x5800f)
    #1 0x4007d0 in main /tmp/smh/boom.cpp:2
    #2 0x7f4181b04d5c in __libc_start_main (/lib64/libc.so.6+0x1ed5c)
```

AddressSanitizer output 2

- Memory layout

```
SUMMARY: AddressSanitizer: heap-buffer-overflow /tmp/smh/boom.cpp:3 main
Shadow bytes around the buggy address:
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fc0: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x0c287fff9fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff9fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c287fff9ff0: 00 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
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
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
```

AddressSanitizer output thoughts

- Detailed
- Colourful
- A bit intimidating?
- Not easy for non-expert developers to interpret
- So far, not as “googlable” as Valgrind errors

AddressSanitizer

- Can find memory errors (more types than Valgrind, see backup) but not uninitialised values
- Fast-ish: only a factor of two slower than running normally
- $\approx$ 30 ATLAS bugs found over $\approx$ 3 months, 16 ROOT bugs found over two years (David Obdurachmanov from CMS)
- Why aren't we running it as a matter of course?
  1. Not ABI-compatible - need to recompile everything, including entire LCG stack (may get fixed in future)
  2. Always crashes on first error
  3. GCC implementation cannot blacklist known issues (clang can)
- 1. indicates a deficiency of our build system (should be easier after CMake migration)
- 2. can be frustrating if the bug is in external code

Undefined behaviour sanitizer

- Looks for various classes of undefined behaviour
  - Signed/unsigned integer overflow (see entertaining GCC discussion)
  - Shift ($<<$ and $>>$) errors
  - Passing wrong value to bool/enum
- Does not stop execution on first error

Undefined behaviour sanitizer

- Crystal-clear error messages (line number, explanation)
  - ALFA_GeometryReader.h:132:16: runtime error: load of value 1551839731, which is not a valid value for type 'eGeoSourceType'
  - HepMcParticleLink.h:72:51: runtime error: left shift of 200001 by 16 places cannot be represented in type 'int'
  - GaudiKernel/ToStream.h:372:25: runtime error: load of value 190, which is not a valid value for type 'bool'
  - BunchGroupSet.cxx:38:23: runtime error: shift exponent 99 is too large for 32-bit type 'int'
  - FlexBinChunk.icc:264:110: runtime error: signed integer overflow: -2147483642 + -2147483648 cannot be represented in type 'int'

Undefined behaviour sanitizer (ubsan)

- UBSan is particularly powerful since there is no other way to catch these kinds of error
- An error from UBSan means one of two things (assuming no bug within UBSan)
  1. The code is calculating nonsense
  2. The code is accidentally working but may break with a new compiler/platform
- UBSan now runs regularly in debug platform for devval

ThreadSanitizer

- Checks for data races
- Reputedly much faster than Helgrind - Valgrind tool for data races
- Tried compiling Intel Threaded Building Blocks and TBB examples with ThreadSanitizer
  - Generate 403 warnings while compiling examples, hangs somewhere
  - Generate 40 warnings using suppression files, compilation still hangs
- May or may not end up being useful for us

Code coverage

- Static analysis tools have 100% code coverage, but do not catch 100% of problems (100% coverage, $\approx$ 10% depth)
- Dynamic analysis tools have test-dependent code coverage, can catch more problems (X% coverage, $\approx$ Y% depth)
- How can we measure code coverage effectively?
- How good is our RTT test coverage?

Wishlist

- **asan** running regularly for a full build of some sort
- **ubsan** running in RTT and continuous integration tests
- **tsan** properly evaluated for AthenaMT
- Working clang build (more sanitizer options)
- Developer education for **ubsan** and **asan**
- More effort available to follow up on issues found

Backup: AddressSanitizer (asan) issue types found

	ASan	Valgrind
Heap out-of-bounds	**Y**	**Y**
Stack out-of-bounds	**Y**	N
Global out-of-bounds	**Y**	N
Use after free	**Y**	**Y**
Use after return	**Y**	N
Uninitialised memory read	N	**Y**
Initialisation order issues	**Y**	N
Leaks	N	**Y**
Slowdown	2x	10x