Runtime verification

(aka Dynamic Analysis)

Yuri Gribov, Samsung Advanced Institute of Technology

Lalambda '21

About the author

- Teamlead @ Samsung Moscow
- Accidentally became a compiler writer 15 years ago
  - In-house, GCC, LLVM, neurocompilers (also some HPC and gamedev)
- Passionate about verification in general and dynamic/static analyses in particular
  - GitHub yugr
  - Habr the_real_yugr

Disclaimer

- A big picture overview without delving into details of particular checkers or technologies
- Engineering focus
- C/C++-focused (although ideas are generally applicable)

This presentation is available at

- (slides) https://github.com/yugr/Lalambda/blob/master/21/talk.md
- (practice) https://github.com/yugr/Lalambda/blob/master/21/practice.md

Overview

- Runtime verification aka dynamic analysis
- Instrumentation of programs to verify behavioral invariants at runtime
  - safety, performance, etc.
  - verifying code is called a "monitor"
- (Much) more widely used in industry than static tools:
  - no false positives
  - no scalability problems
  - reprocases easily available

Disadvantages

- Limited coverage
- Solved via
  - fuzzing
  - rule/grammar-based input generators
  - A/B testing (in production environments like Google services)

Example analyses

- Virtual memory :)
- Sanity checks in code
  - e.g. C/C++ assertions in programs
  - e.g. Glibc `malloc` or libstdc++ iterator internal checks
- Valgrind
- Sanitizers (Asan, UBsan, Msan, Tsan, etc.)
- "Business rules" (GDPR, data minimization, etc.)

Community

- Academia (Runtime Verification conference)
  - grew out of model checking in 2000-s (Runtime Verification - 17 Years Later)
  - verify complex modal logic formulas on program traces
  - usually applied to interesting but niche projects
- Industry (hackers and corporations)
  - automatically detect bugs at large scale (without manual work by user)
  - much older (malloc debuggers existed at least since 80-s)
  - typically much more influential

Dynamic analysis algorithm

```
errors = {}
program_with_monitor = instrument(program, spec)
while test_corpus not empty:
  test_input = test_corpus.pop()
  errors, coverage, ... += program_with_monitor(test_input)
  update test_corpus
```

Dynamic analysis algorithm

```
errors = {}
program_with_monitor = >>>instrument<<<(program, >>>spec<<<)
while test_corpus not empty:
  test_input = test_corpus.pop()
  errors, coverage, ... += program_with_monitor(test_input)
  >>>update test_corpus<<<
```

Ontology of dynamic analysis project

Runtime analysis tool ("checker") contains of three main "parts":

- spec: an invariant that we want to check
- instrumentation (aka monitor): a way to verify that invariant is preserved during execution
- test corpus: input data which we run the checker through

New successful checkers are created by innovating in any of the three components.

Creating new checkers: spec

The "spec" part:

- come up with a new interesting class of bugs and propose method to autodetect them
- most interesting classes already handled :(
- e.g. Sortchecker was the first tool to check qsort axioms

Spec taxonomy (1)

- Memory errors (Asan/Msan, Valgrind):
  - liveness errors: accessing after end-of-life (use-after-free, use-after-return, iterator invalidation)
  - buffer overflow: heap, global, stack
  - uninitialized memory
  - memory leaks
- Typing errors (in non-type safe languages like C)
  - aliasing violations (TypeSanitizer)
  - mismatched types (libcrunch)

Spec taxonomy (2)

- Parallel programming errors (Tsan):
  - deadlocks and data races
- Language-specific errors:
  - integer overflows (UBsan)
  - static init order fiasco (Asan)

Spec taxonomy (3)

- Invalid usage of APIs:
  - not checking return codes of syscalls or standard APIs
  - mismatched memory allocation API (calling `free` on `new`-ed pointer)
  - invalid comparators

Spec taxonomy (4)

- Violation of "business rules":
  - very application specific
  - specifications are extracted from domain experts, architects, QA, etc.
  - Runtime Verification, from Theory to Practice and Back and Industrial Experiences with Runtime Verification

Creating new checkers: instrumentation

The "instrumentation" part:

- for an existing spec, develop new ways to detect more errors more efficiently
- often determine whether checker will be used
- e.g. there were many buffer overflow checkers before AddressSanitizer but too slow or with limited coverage

Instrumentation taxonomy

- Aka aspect-oriented programming (AOP)
- Runtime verification is trivial in languages like Python or Java
  - full access to AST at runtime
  - many AOP frameworks
- Instrumentations of native code are categorized by stage in compilation pipeline and mechanism used for instrumentation
  - compromise between simplicity of implementation/integration and desired level of detail

Instrumentation taxonomy: preprocess-time

- Code can be instrumented by forced inclusion of debug header
  - e.g. via `-include mychecker.h`
  - header would contain something like
    `#define malloc my_safe_malloc`
- Examples:
  - dmalloc
  - Glibc _FORTIFY_SOURCE

Instrumentation taxonomy: compile-time

- Compile-time instrumentation:
  - source-to-source (e.g. libcrunch)
    - traditionally done via CIL but it's C only :(
    - Clang LibTooling supports C++ but is complicated to use due to baroque AST
  - codegen-based (e.g. Asan or DirtyPad)
  - asm-based (e.g. AFL or DirtyFrame)

Instrumentation taxonomy: link-time

Link-time instrumentation:

- replacing normal code with "checking" implementations at link time
- e.g. via `-Wl,--defsym,malloc=my_safe_malloc` or `-Wl,--wrap=malloc`
- e.g. [_malloc_dbg]() replaces normal `malloc` if user links against debug version of Microsoft runtime

Instrumentation taxonomy: run-time

Run-time instrumentation types:

- `LD_PRELOAD`-based (e.g. ElectricFence, sortchecker, failing-malloc)
  - `LD_PRELOAD` is a canonical way to implement AOP on Linux
- syscall instrumentation (e.g. SystemTap)
- dynamic binary instrumentation (aka DBI, e.g. Valgrind, DynamoRIO or Intel Pin)

Creating new checkers: test corpus

- Project testsuites provide insufficient code coverage
- Need to extend test corpus by generating new tests:
  - via fuzzing:
    - random (e.g. Radamsa, zzuf)
    - feedback-driven (e.g. AFL, libFuzzer)
    - concolic (e.g. Microsoft SAGE, Mayhem, KLEE)
  - by developing generator for sufficiently important class of data
    - e.g. Defensics supports grammar-based test generation for 250+ protocols
    - e.g. Csmith generates random C++ code for compiler testing

How to test a checker

- Once checker is ready you'll want to test it on as much code as you can
- Try to apply it to important OSS projects
  - archivers, media processing libraries, browsers, etc.
  - to find interesting package faster:
    - package popularity rating
    - Debian codesearch (supports both web and cmdline interfaces)

How to test an LD_PRELOAD- or DBI-based checker

- Checkers which do not require program recompilation are easier to test:
  - Run all apps in `/bin` and `/usr/bin`
    - without params, with `--help`, with `--version`
    - automatic but coverage is low (tests initialization code, at best)
  - boot complete Linux distro with your checker preloaded
    - for example valgrind-preload
    - limited applicability
    - need to perform manual actions to explore system behavior

How to test an arbitrary checker

- Run package unittests (if available)
  - good coverage but not scalable (5-30 minutes per package)
  - tiresome and demotivating :(
- System testsuites
  - run system benchmarks (e.g. Phoronix suite or browser testsuites)
- Instrument complete Linux distro (e.g. sanitize Tizen)
  - extremely hard...

How to test a checker: comparison

Test	Automatic	Coverage	All checkers
Running apps with standard params	Y	Low	Only LD_PRELOAD/DBI
System testsuites	Y	Average	Y
Manual package testing	N	High	Y
Distro boot	iff LD_PRELOAD/DBI	Average (need manual actions to increase)	Y

Using distro build systems

- Linux distros come with a vast number of packages
- Distro build systems can be reused
  - to apply checkers under the hood
  - and run package-specific unittests
- Debian build toolchain
  - Sadly only builds, not tests, but ...

debian_pkg_test

- With some hacking we can make Debian build system to run unittests for us!
- [debian_pkg_test]() project
  - based on [pbuilder]()
  - runs `make check` (or other standard test commands) once package build completes

Trends (1)

Increasing fuzzing speed and efficiency (coverage) by various means

- feedback-driven ("grey-box")
  - AFL and related tools (gofuzz, libfuzzer, etc.)
- symexec-driven ("white-box")
  - Billions and Billions of Constraints: Whitebox Fuzz Testing in Production
- various combinations thereof

Trends (2)

Increasing fuzzing adoption in community:

- integration of fuzzers into development lifecycles (kudos to @msh_smlv)
- inspire project owners to write fuzzing for their projects through initiatives like [OSS-fuzz]
- bug bounty programs e.g. [Google Fuzzilli]

Links

- Runtime Verification conference (Springer)
  - Too scientific
  - Most papers are on verifying temporal logic assertions at runtime
- More practical: vulnerability reports
  - CVE reports
  - DEFCON
  - Blackhat
  - Phrack

Advertisement

Samsung System-On-Chip team is hiring developers
to develop state-of-the-art compilers in Moscow Research Center:

- NPU Compiler Developer for Exynos AI Accelerator (https://hh.ru/vacancy/42341825)

Also need GPU performance engineers, whatever that means (https://hh.ru/vacancy/44907512).

The End

Please share your ideas on runtime verification (new checkers, novel ways to test them, etc.):

- tetra2005 beim gmail punct com
- TG the_real_yugr
- GH yugr

Checker gotchas

- Instead of testing that bad objects are not accessed, make sure that such accesses cause havoc
  - Fill undef memory/regs with garbage (MSVS does this with mallocked memory)
  - Unmap page after buffer to force segfault (ElectricFence)
  - Fill gaps in stack frame with random values (DirtyFrame)
  - Fill struct pads with random values (DirtyPad)
  - Intro random delays in Pthread-based programs