Design and development of Svace static analyzers

Andrey Belevantsev, Alexey Borodin, Irina Dudina, Valery Ignatiev, Alexey Izbyshev, Sergey Polyakov, Evgeny Velesevich, and Dmitry Zhurikhin
Ivannikov Institute for System Programming of RAS

Ivannikov Memorial Workshop 2018, Yerevan, Armenia

Agenda

- Static analysis and development lifecycle
- Haven’t we solved the problem yet?
- Design decisions and lessons learned*
  - Overall architecture
  - Supporting infrastructure
  - Analysis organization
  - Analysis algorithms
  - Warning review
- What lies ahead

* See also:
Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53, 2 (February 2010), 66-75.

Static Analysis Requirements

ISP RAS

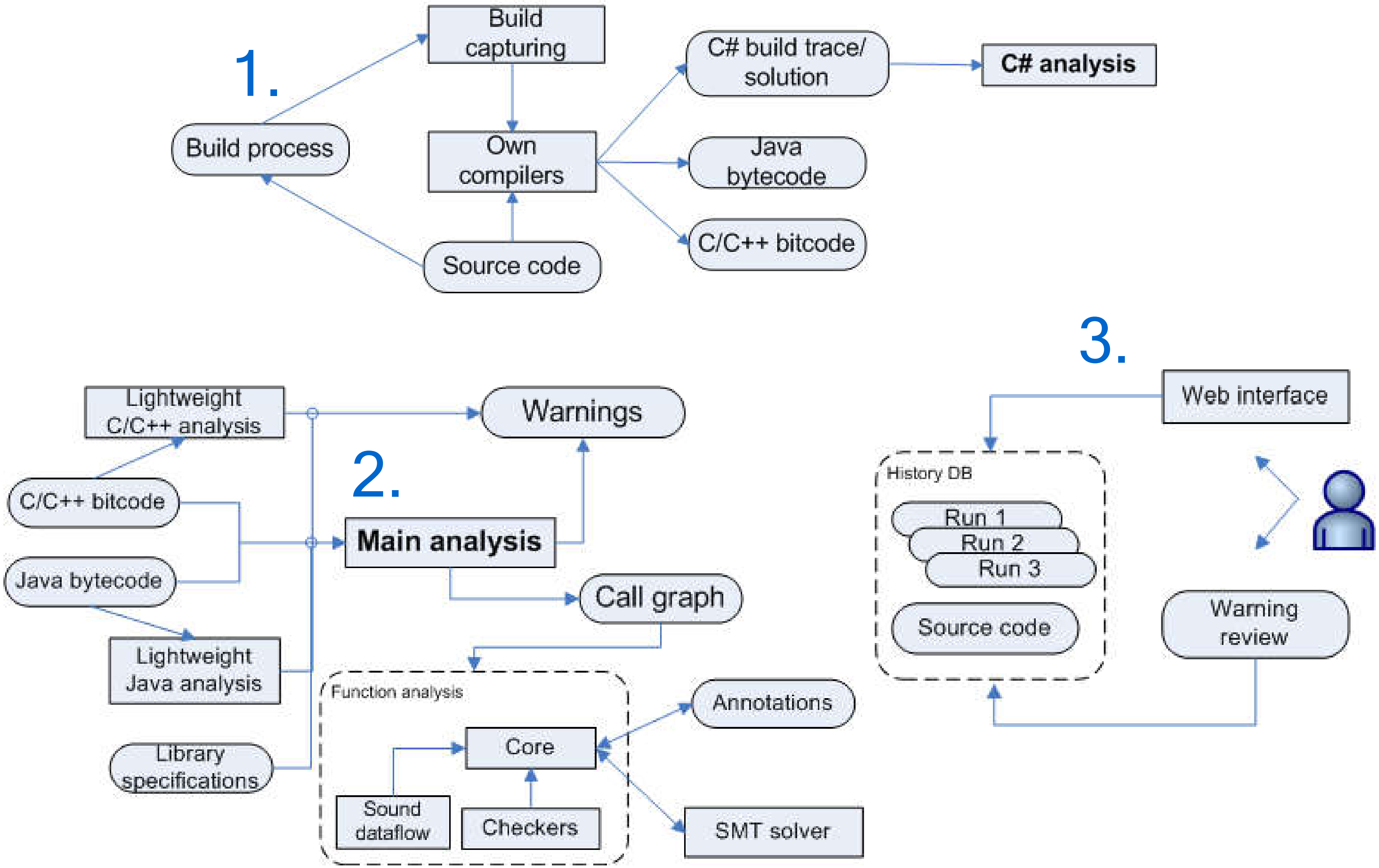
- Wide applicability: defect detection, program understanding, performance, ...
- Application for secure development lifecycle
  - CI integration, nightly builds, Q&A
- Requirements that follow:
  - Fully automatic analysis (no need to change the code)
  - Scalable to millions of LOC
  - Fair percent of true positives (*60+%*)
  - Sufficient *completeness**
  - Support of programming languages (C/C++/Java/C#), defect types (many), environments (Windows/Linux)
  - Extensibility with new checkers, flexibility (tailored config)

(Scalable to millions of LOC, fair percent of true positives, sufficient completeness: Quality tradeoff)

Haven't we solved it yet?

- A few production tools – can't they just evolve?
- Some non-trivial issues come to mind:
  - Difficult to compare with competitors: their licenses won't allow
  - Difficult to choose the evolution directions
    - memory modeling (separation logic etc.)
    - conventional analyses become demand-driven
    - SMT solvers become more powerful and change the way analyzers build queries for errors detection
  - The goal of saving developer time is taken further
    - *Tightly* integrate with developer's workflow (CI)
    - Prioritize analyzer's output so that true warnings are seen first
    - Suggest and make *semi*-automatic code fixes

Svace Architecture



Build Interception

- Detect process launch
  - LD_PRELOAD to dynamically linked executables
  - Debugging API (`ptrace`, WinAPI)
  - Wrappers (e.g. MS-DOS machine within Windows)
  - Java: agent injection for compilation APIs interception
  - C#: msbuild DLL injection (similar to Java)
- Parse cmdline/environment
  - Trace “interesting” launches
  - Decide on action (usually – run own compiler)
  - Transform cmdline (options/envvars) for our compiler, not loosing significant options, include paths, ...
- Launch our compiler for generating IR (or other needed tools)

Constructing An Analyzer Compiler

- Harsh requirements
  - Need to be as failproof as possible
  - Need to understand C/C++ dialects of dozens of desktop/embedded compilers
  - Need to understand modern language standards
- Has to base on production open source (C/C++ → GCC/LLVM) or buy EDG
  - Add some “fuzzy parsing” mechanism (ie not stop on error, but recover as much as possible)
  - Fixup for dialects (or “morph” user source to get rid of them)
  - Inject additional data if needed by the analyzer
  - >1000 patches wrt vanilla Clang
- Java/C# is no problem (one compiler)
  - Though Google invented (and deprecated) Jack compiler...

Analysis Engine

- Extensibility
  - Need to support many warning types / checkers
  - Ways to reuse code and calculated data for checkers (effectively the data that is always required becomes “core”)
- Multiple language support
  - Lower level common IR
  - Ensure that analysis assumptions are honored when making IR
- Call graph reconstruction
  - C/C++: requires gathering linkage information
  - C++/Java/C#: requires (some) devirtualization
- Parallel analysis
  - Analyze in parallel independent call graph parts
  - Take into account function locality w.r.t. modules
  - Speed / memory consumption tradeoff

Analysis Engine – II

ISP RAS

- Incremental analysis
  - May be used for CI integration and for running on developer’s PC
  - Needs changes in all components (“merging” old and new data)
  - Need to understand whether to draw a line in analyzing the unchanged code with changes in context
- Determinism
  - Same/slightly changed results for same/slightly changed source
  - Varying input data (due to build issues)
  - Analysis results grouping
  - Dependence on the iteration order over input data (fixup the order or change the algorithm)
  - Function analysis timeouts (avoid “real” timeouts if possible)
  - Statistical checkers

Analysis Algorithms

- Memory model / aliasing
  - Field sensitivity, limited number of dereferences
  - Alias analysis, escaped memory analysis, strong/weak updates
- Sound / unsound
  - Most analysis is unsound (parameter aliases, loops, limitations on summary / derefs)
  - But need fully sound part (unreachable code, functions exiting program)
- Tracking values’ properties
  - Reason about properties of values, not of memory cells
  - Should be careful when there are multiple ways to reinterpret the value

Analysis Algorithms – II

ISPRAS

Parameterized summaries

- Scalability requires limiting the number of passes over a function
- Context sensitivity means varying analysis behavior for different calling contexts
- Using function summaries that parameterize analysis results on external memory means visiting every function only once
- Careful to put to the summary only the data describing “escaped” memory and to limit its size
- Checkers should decide what information is important to save

“Symbolic” external values

- When treating unknown values’ properties conservatively, intraprocedural analysis doesn’t yield anything useful
- Try saving any merges/*computations* with such values and resolve them upfront in the call graph with concrete values

Error Definitions

- Beliefs / inconsistencies
  - Known way to detect errors: find inconsistencies in the assumptions the code does about some dataflow facts
- Find a control flow "segment" where:
  - either there's always an error if we go there, or
  - the "segment" is unreachable / unfeasible
  - Both ways it makes sense to warn
  - Rely on *programmers* not writing unneeded code
- A segment may be a control flow edge or an execution path (for path sensitive analysis)
- Statistical checkers also find "inconsistencies" (*mostly* done this way → this way is "right")
- Language specific definitions are possible (C++, Java)

Path sensitivity via symbolic execution

ISP RAS

- ❑Core engine computes a path predicate
- ❑Symbolic states are tracked and merged
- ❑Checkers are free to attach conditions to the attributes they are tracking
  - ➢We are still reasoning about values, not memory
- ❑Just tracking values' changes (aka "symbolic state") is often not enough
  - ➢Taking a specific path is useful information (comparisons)
- ❑Condition simplification
  - ➢It is useful to apply a handful of trivial simplifications before passing a query to an SMT solver

Warning Review

- Convenient review interface
  - Web-based, but now better be integrated into CI or dev.env
  - “Dashboard” (bird’s eye view)
- Runs comparison
  - Never see anything once reviewed as a false positive
  - Need to support slightly changed code and repo branches
- Lots of data to store (except analysis results)
  - Source code to show it
  - Tokens/relations to do syntax coloring/navigation
  - Even more for e.g. incremental analysis
- Migration without loosing review
  - Match warnings between releases to avoid spurious new stuff
  - Avoid too much churn (cf. mentioned Coverity paper)

What Lies Ahead – Analyzer

ISP RAS

- More of aliasing
- Better loop handling
- C++/Java collections
  - No chance to infer their semantics through implementation, need to make their primitive operations “first class” in the IR
- Serving to different clients
  - Basic use case assumes warnings will be reviewed by humans
  - Need to configure “verboseness”
  - Need to adapt to the dynamic analysis toolchain use case
- Analysis API
  - Some checkers (e.g. simple source-sink ones) can be done by customers, and they wish to do so
  - External API will help but needs resources for support

What Lies Ahead – Around Analyzer

- Being input to further analyses
- Prioritization
  - Sort out warnings to make true ones stand out
- Code fixes
  - Suggest fixes to certain warnings and (optionally) apply them
- Improving review experience (ML)
  - Direct more attention to the changes that are potentially risky
- Code base wide refactorings
  - Making changes guided by static analysis results for the whole code base (think hundreds of git repos)

ISP RAS

Success is a team effort