Static analysis techniques in security software development lifecycle: requirements, problems, features

Andrey Belevantsev
Leading Researcher, ISP RAS
abel@ispras.ru

Agenda

- Static analysis industrial requirements
- Svace architecture
- Problems to solve
  - Infrastructure (build interception, compatibility, parser, ...)
  - Analysis (IR, core design, interprocedural, path sensitive, ...)
  - Warning review
  - Multiple levels/languages of analysis
- Research directions
- Conclusions

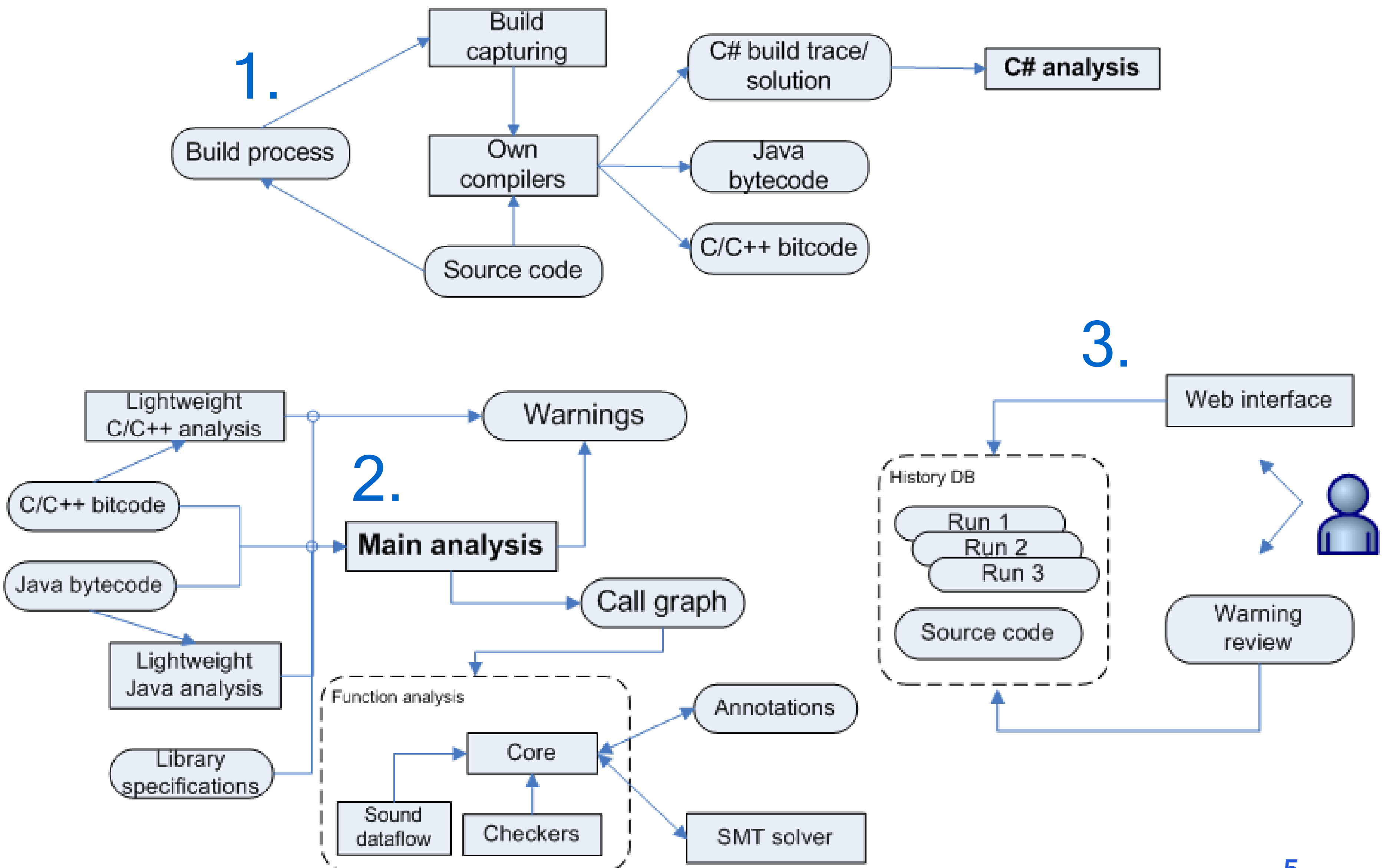
Message of the talk

- Static analysis: an innovative technology requiring many efforts for successful production deployment
  - Many research problems, from fundamental to industrial research
  - Many tasks to solve that do not follow from research, but only from customer feedback
- Static analysis: a technology requiring constant research to stay within or ahead state-of-the-art

Static Analysis Requirements

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

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 Own 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)
  - But then Google invented Jack compiler for Android...

Environment Support

- Build your tools on all supported hosts
  - Various Windows flavors (mostly fine but WinAPI differences can be trouble)
  - Various Linux distributions (hello kernel version 2.4)
  - Some tools should work under harsh restrictions (e.g. chroot system)
- Avoid conflicts with system tools
- Provide enough logging capabilities for fixing issues reported by a customer
  - Usually both customer environment and source code is not available
  - Need to direct 1st line of support to get required data

Analysis: Intermediate Representation

- Multiple analysis levels
  - AST-level checkers are usually language specific and performed within corresponding compiler environments
  - Clang Static Analyzer, FindBugs, Roslyn, ...
- Main analysis intermediate representation
  - Capable of presenting several languages (C/C++/Java)
  - Tradeoffs: somewhat high level (closer to rich AST) ...
    - Harder analysis (many node types) but no problem with source code connection
  - ... or somewhat lower level (closer to bitcode, LLVM IR)
    - Easier analysis but need good debug information (issues with reconstructing types, names, ...)
  - May be lured to the IR chosen by your compiler

Extensibility

- Need to support many warning types (dozens) and many checkers (hundreds)
- Design the analysis engine so that it would be easy to extend
  - Core part: compute program information (call graph, control flow, data flow) needed by most checkers
    - When made right, adding a new checker wouldn't slow down the analyzer (much)
  - Checkers part: plugins caring for specific "situations" in source code that look like a certain type of error
    - May have many checkers detecting the same error type (with different confidence, approach, limitations, etc)
    - Checkers calculate some special data ("attributes") based on the core engine information

Extensibility - II

- Typical data to put into core
  - Memory model and alias analysis
  - Value reasoning (akin to numbering)
  - Interprocedural handling (separate slide)
  - Conditions tracking for path sensitivity (e.g. conditions necessary for the execution to reach the current program point)
- Multiple levels of checkers are also present in the main engine
  - Not all checkers need everything the core part computes
  - Should be possible to differentiate based on checker rqs
- Main engine is generally unsound
  - But need a part to compute sound (conservative) dataflow information to rely on (e.g. unreachable code)

Interprocedural Issues

- Need to select the basic design for interprocedural analysis
  - Resume / annotation - based (most popular choice)
  - Inlining based (limited scalability)
- Issues to solve
  - What to put in function annotations
  - How to limit the amount of data
  - Any limitations should be dependent on the core data computed, not checkers
    - Otherwise enabling/disabling a checker may lead to change in reported warnings for an unrelated checker

Path Sensitivity

- Various degrees of freedom
  - Way to represent the conditions (e.g. we allow conjunction / disjunction, but negation is allowed only on atoms)
  - Which SMT solver to use (Z3 is the usual choice)
  - Whether the conditions should be (somewhat) simplified or fed to the solver as is (we make some easy ones)
- Changes in the interprocedural support
  - Limit on the boolean formula length that can be put in the annotation
  - Policy on shorting the formula (making it more rough by replacing some parts with true constant)

Linking Tracking Support

- Analyzer needs to distinguish between program components when processing a complex system (e.g. Android)
- For C/C++, take this data from the linking info (knowledge what got linked into where)
- Allows analyzer to:
  - Properly connect functions when building a call graph (when having multiple choice for a external function, sometimes just choosing heuristically is not enough)
  - Analyze by component and throw away data calculated for internal functions

Scalability

- Parts of call graph can be analyzed in parallel
  - Strive for maximum “breadth” within call graph
  - When reading a module, schedule for analysis a function from another already read one
  - When a module is fully read, try to process functions within it as much as possible while they are in memory
- Load balancing
  - Find a trade off between amount of parallel work and consumed memory
  - Coordinate between different analyzers working simultaneously on the host

Determinism

- Users want to see the same set of warnings from each analysis run of the same source (or slightly different source)
  - Even if the source was built several times
  - Reason is to avoid spurious new/removed warnings during warning review process
- Not easy to achieve this in a large system
  - Analyzer has various limits to avoid extreme complexity for corner cases and large functions
  - Limits should be chosen carefully being not dependent on checkers, only on core data
  - Any decisions the analyzer makes should not be based on possibly varying data between builds

Other specifics

- Multiple language support
  - With lower level IR some higher concepts (templates, exceptions, etc.) are already lowered by the compiler
  - Need to recover them carefully
  - Basic algorithms baked into the core part should work well for all supported languages
  - Avoid language specific heuristics in the analyzer
- Incremental / remote analysis
  - Separate use cases that require support in all tool parts (build interception, analysis, results handling)
  - Merging analysis data of the newly changed part with the main analysis data can be tricky

Warning Review

- Database of analysis runs
  - Should be able to hold a number of analysis results, source code analyzed
  - Should be able to compare arbitrary runs
- Basic requirement: hide any warning that was reviewed once as a false positive
- User interface
  - Web-based interface - a popular choice
  - IDE integration
  - “Dashboard” (manager data)
  - Not possible to build without deployment and real customer feedback

Future Research

- Constant research within and around the main analysis technology
  - Most ideas do not get into the product, but it is the only way to maintain competitive technology level
- Main engine tasks
  - Better memory model (alias analysis)
  - Better call graph construction (devirtualization)
  - Loop analysis
  - A subsystem for popular kind of taint-based checkers
  - A user API or a DSL for such a subsystem

Future Research - II

- Analysis approaches that are different enough from mainstream
  - E.g. separation logic allows to have precise shape analysis for dynamic memory (Infer tool)
  - E.g. searching for code clones of known true positives
- Automatic code fixes / suggestions (not easy for non-trivial checkers)
- Applying machine learning techniques
  - Warning prioritization
  - Fixes suggestion
  - Statistical checkers (already present in production tools)
- And more ...

Message of the talk

- Static analysis:
  - an innovative technology requiring many efforts for successful deployment
  - a technology requiring constant research to stay within or ahead state-of-the-art
- For success you need:
  - An experienced large enough team
  - Feedback from industrial partner
  - Many years of work (started research in 2002, started productization in 2009, deployed in 2015)

Thank You