Software Security 05

Sven Schäge

Methods for Finding Security Vulnerabilities

Sven Schäge

The Complementary 3

- There are three main methods to find security vulnerabilities in practice
  - Static Code Analysis
    - At compile time
  - Fuzzing (smart generation of tests and subsequent testing of software)
    - At runtime
  - Review by experts (penetration testing)
    - Formal Verification?
- Each method has benefits and drawbacks; however, they complement each other nicely
- Nevertheless, they cannot guarantee the absence of security vulnerabilities

Static Code Analysis, Static Application Security Testing (SAST)

- + Automatic
  - Most of the work is done by tools that automatically scan the code
  - Scales well
- + Computationally cheap
- + Does not need the software to be run
- - Can raise false alarms
  - Not guaranteed to find all bugs – this depends on the specification of bug
- - Cannot find bugs that only reveal themselves when software is running in complex runtime environments
- + Cover the entire code even parts that will be executed only in rare cases
- - Cannot be used to find errors in runtime environment
- + Finding responsible code positions is easy

Fuzzing and Dynamic Application Security Testing (DAST)

- + Automatic
  - Scales well
- + Cannot raise false alarms
- - Computationally more expensive
  - May scan year-round
- - Needs the software to be run (dynamic analysis)
- - Not guaranteed to find all bugs
- + Can find security vulnerabilities in deployed environments
- + Does not require the code
- - Testing may launch an attack!
  - Testing should be performed in production-like but non-production environment.
- - Cannot cover all of the source code and thus application
- + Can find errors in runtime environment
- + Can be used to verify results of Static Code Analysis
- - After finding a vulnerability, we have to find the code position that is responsible for it

Review by Experts

- - Manual inspection with focus on security
  - Does not scale well
- - Requires time and can be (comparably) expensive
- + No false alarms
- - Not guaranteed to find all bugs

Important Quality Metrics

Completeness and Soundness of Analysis

- When classifying programs, it depends on the mandate (the property that programs should be classified by) used whether the classifier can be called complete or sound.
- In this lecture we will restrict ourselves to the following understanding of completeness and soundness **unless the mandate is not explicitly stated**.
  - If we say a classifier/program checker/analysis tool is complete we mean that it will classify all programs that have bugs as having bugs. (Minimizes false negatives, ensuring that the tool does not miss real vulnerabilities)
  - If we say that a classifier/program checker/analysis tool is sound we mean that it will always classify programs that are bug-free as bug-free. Soundness in software security refers to the assurance that a security tool correctly identifies all actual security vulnerabilities without reporting false positives.
  - This means we use the mandate **buggyness**, trying to answer the question „Is the program buggy?“
- In general, inverting the mandate of a classifier will swap the results with respect to soundness and completeness!

Measuring Accuracy: Precision and Recall for **Finding Buggy Programs**

the terms in the table are buildt as
X Y where
X∈ {True, False} indicates correct or incorrect classification and
Y∈ {Positive, Negative} indicates the result of the Analysis

program is rightfully declared to contain a bug

buggy programs that have not been caught

		Analysis Outcome (Marked as having no security weakness)	
		Alarm	No Alarm
Program Ground Truth	Buggy	True Positive	False Negatives
	Bug-free	False Positive	True Negative

false alarms

program is rightfully declared to not contain a bug

Completeness and Soundness of Analysis

- For our purposes, (perfect) completeness means if a program is claimed to be buggy then it always is buggy.
- (Perfect) Soundness means that if a programs is claimed to be buggy the it is with buggy will always be identified as buggy.
- Precision and Recall quantify these absolute categories
- Precision
    - Of the items flagged as buggy, how many are truly buggy?“
    - Precision=True Positives/(False Positives+True Positives)
    - No False Positives (no false alarm)=>Precision=1
    - 100 security alerts but only 90 of them are true positives it means that we have a precision of 90%
- Recall:
    - Of all the truly buggy items, how many did the system correctly identify?“
    - Recall=True Positives/(True Positives+False Negatives)
    - No False Negatives (all bugs caught) => Recall=1
- **False Positives (FP):** Items that are incorrectly classified as buggy when they are not. (false alarm)
- **False Negatives (FN):** Items that are buggy but are not identified as such by the security tool. (bug that is not caught)
- **True Positives (TP):** Items that are correctly identified as buggy.
- **True Negatives (TN):** Items that are correctly identified as bug-free.

F-Measure

- The F-measure is a standard measure of accuracy, combining precision and recall
- Harmonic mean of recall and precision
- Hypothetical, ideal analysis: F-measure=1
- Here precision and recall are equally important

F-measure = 2 / ((1/precision) + (1/recall))
=2*recall*precision/(recall+precision)

Fx-Measure

- Depending on the applications we may want to weigh precision and recall differently
- The Fx-measure accounts for this using parameter x
- The Fx-measure was derived so that it measures the effectiveness with respect to a user who attaches x times as much importance to recall as precision
- Two commonly used values for x are 2, which weighs recall higher than precision, and 0.5, which weighs recall lower than precision.

Fx-measure=(1+x^2)*recall*precision/(recall+ (x^2 * precision))

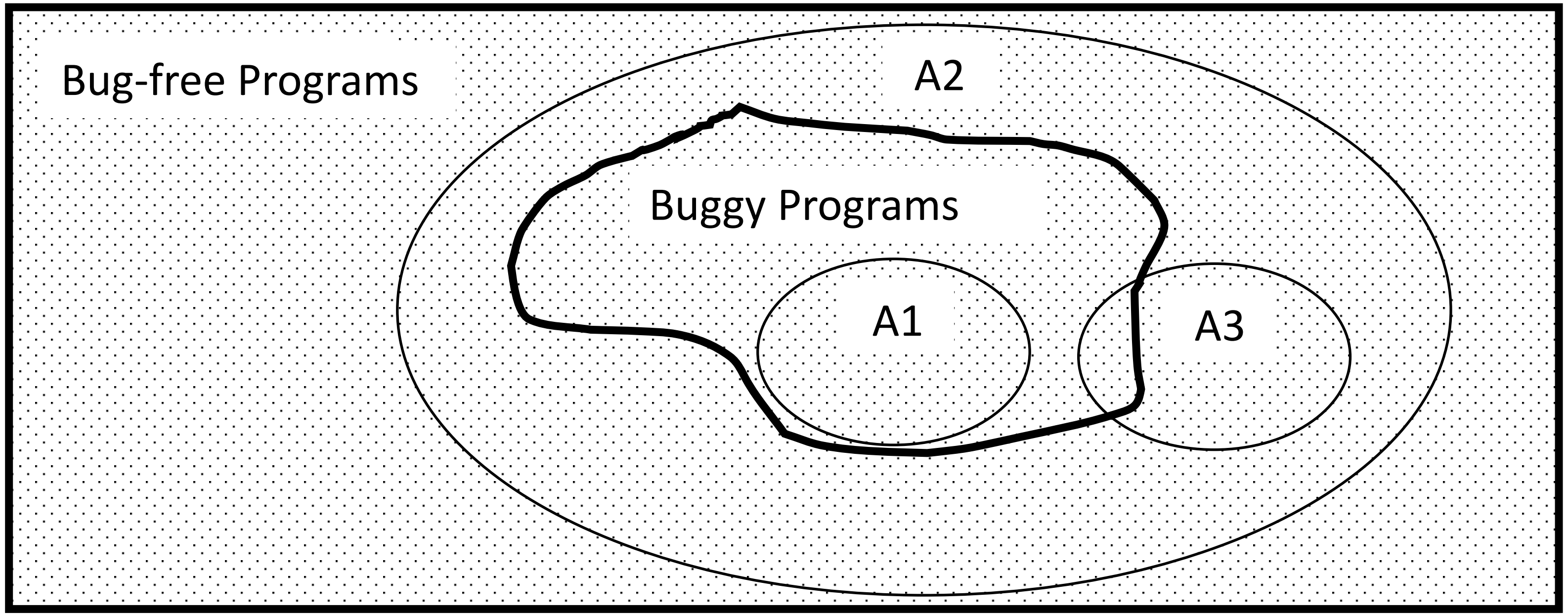
Application-Aware Choice of Analysis

Choosing an Appropriate Program Analysis: Completeness vs. Soundness

- Assume the analysis tools A1, A2, A3 recognize their respective inner spaces as buggy programs, their outer space as bug-free programs. Assume their mandate is to find buggy programs.

	Complete?	Sound?
A1	Yes	No
A2	No	Yes
A3	No	No

Space of all Programs



On Specifications and Properties

Analysis and Specification

- Usually software analysis depends on a specification of some desirable property of a program
  - These specifications can vary in formality
  - More formal specifications are better for automatic testing
- For example, if we want to avoid division by zero, we could define a simple state machine to capture this as a specification.

Classifications of Specifications

- History-based specification
  - behavior based on system histories
  - assertions are interpreted over time
- State-based specification
  - behavior based on system states
  - series of sequential steps, (e.g. a financial transaction)
- Transition-based specification
  - behavior based on transitions from state-to-state of the system
- Functional specification
  - specify a system as a structure of mathematical functions
- Operational Specification
  - e.g. Petrinets, Algebras

Classification of Properties

- Safety Properties
  - Program will never reach a bad state
  - Examples:
    - assertions
      - predicate at point in program that should always evaluate to true
    - types and type system
      - define how variables should be interpreted, e.g. int
    - pre- and post conditions
      - a precondition is a condition or predicate that must always be true just prior to the execution of some section of code
      - a postcondition is a condition or predicate that must always be true just after the execution of some section of code
    - loop and class invariants
- Liveness Properties
  - Program will eventually reach a bad state
  - Examples:
    - program termination,
    - starvation freedom

Methods for Finding Security Vulnerabilities – Static Code Analysis

Sven Schäge

What can we Hope to Achieve?

- Halting Problem and Rice's Theorem
  There is no algorithm that inputs any program and decides if the program satisfies a given non-trivial semantic property.

- Wishlist
  - Complete (no uncaught bugs)
  - Sound (no false alarams)
  - Automatic
  - Powerful/Non-trivial
- => We must give up at least one point on the wishlist!

Static Code Analysis, Static Application Security Testing (SAST)

- + Automatic
  - Most of the work is done by tools that automatically scan the code
  - Scales well
- + Computationally cheap
- + Does not need the software to be run
- - Can raise false alarms
- - Not guaranteed to find all bugs
- - Cannot find bugs that only reveal themselves when software is running in complex runtime environments
- + Cover the entire code even parts that will be executed in extreme cases
- - Cannot be used to find errors in runtime environment
- + Finding responsible code positions is easy

Static Code Analysis

- Mainly, analyse source code
- Often concentrates on syntactic information
  - Easier to analyse automatically
- Search for patterns or code fragments that meet some rule
  - e.g. every pointer that has been given memory needs to be freed
- Several degrees of locality
  - Consider only single statement at hand
  - or context in entire code
- In general, help the developer to get additional information and new perspectives on the code often in the form of:
  - at position X in the control flow graph we have the property Y

Concepts and Techniques for Static Code Analysis

- Formalization and Visualizations (of code dependencies)
  - Control Flow Analysis (Call Graphs and Control Flow Graphs)
- Iterative Information Generation
  - Data Flow Analysis
- Pattern checking in code for compliance with coding rules
- Type Systems
  - Most widespread form of static analysis
  - Helps to avoid that variables are combined in a way that makes no sense: e.g. add int and char
- Instrumentation of binary code
  - allow for better analysis of program behaviour
  - and trace back origin of problems in source code
  - May add additional checks to the binary that can be evaluated when running the software
- Strong overlap of techniques with techniques used to build compilers

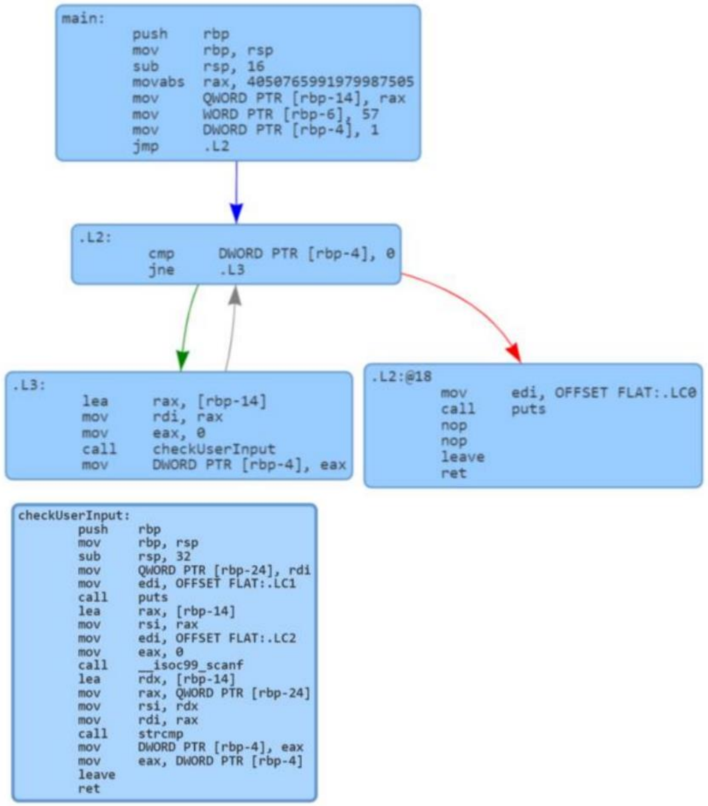
Call Graphs (CG)

- Graph representation of all function calls that a program could make
  - Each node represents a function
  - Edges represent function calls
- Calling relationships between subroutines in a program
- Inter-procedural view of program

Control Flow Graphs (CFG)

- Graph representation of all paths that can/might be traversed through a function during its execution
  - Each node represents a basic block, i.e. straight-line code fragment without any jumps or jump targets
  - Jump targets start a block, jumps end a block
- Generation of Control Flow Graph
  - Starting from Full Flow Graph (every instruction is represented by a node) via edge contraction
- Intra-procedural view of program

Graph output



Prerequisite: The WHILE Language

```
x = 5;
y = 1;
while (x != 1) {
  y = x * y;
  x = x - 1
}
```

(statement) S ::= x = a | S1 ; S2 |
if (b) { S1 } else { S2 } |
while (b) { S1 }

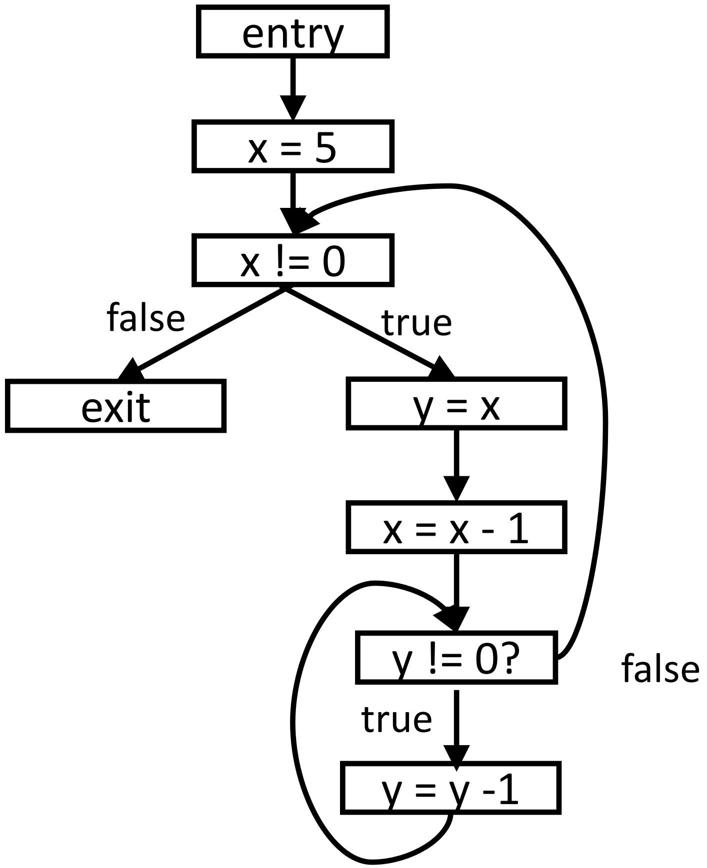
(arithmetic expression) a ::= x | n | a1 * a2 | a1 - a2

(boolean expression) b ::= true | !b | b1 && b2 |
a1 != a2

(integer variable) x

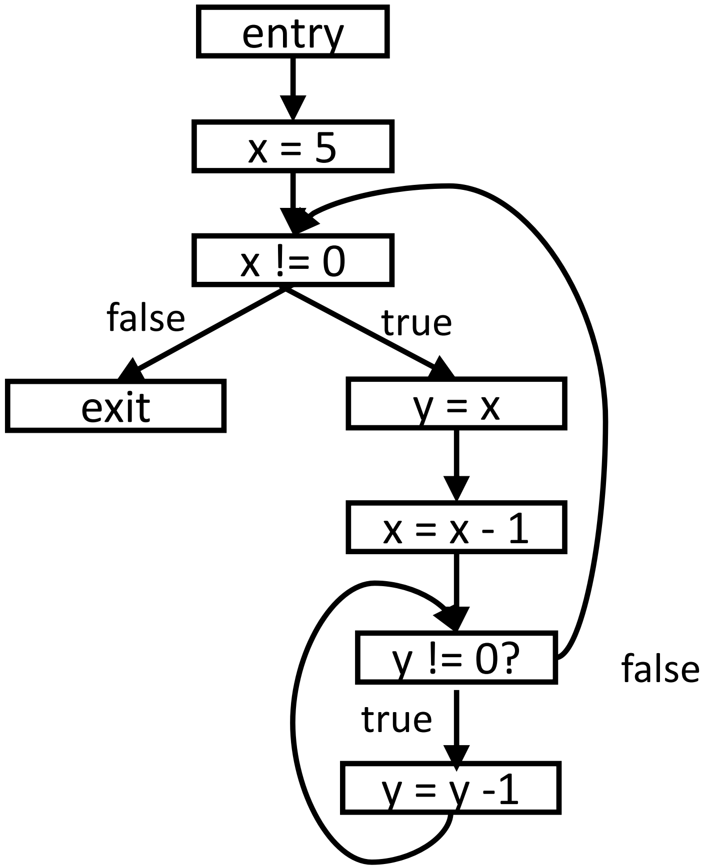
(integer constant) n

Control-Flow Graph



Source of Slide: Mayur Naik

Control-Flow Graph




```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

Source of Slide: Mayur Naik

Inter-Procedural Control Flow Graphs

- Inter-procedural Control Flow Graph
- While CFGs represent the control flow of a single procedure, Inter-Procedural Control Flow Graphs (ICFG) represent the control flow of entire programs
- Combination of CG and CFG

Static vs. Dynamic Call Graphs

- Static Call Graph (undecidable problem -> approximation only)
  - A static call graph is a call graph intended to represent every possible run of the program.
  - The exact static call graph is an undecidable problem, so static call graph algorithms are generally overapproximations.
  - That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.
- Dynamic Call Graph (what parts of the code have been executed in a specific run)

Pattern Checking in Code for Compliance with Coding Rules/Policies

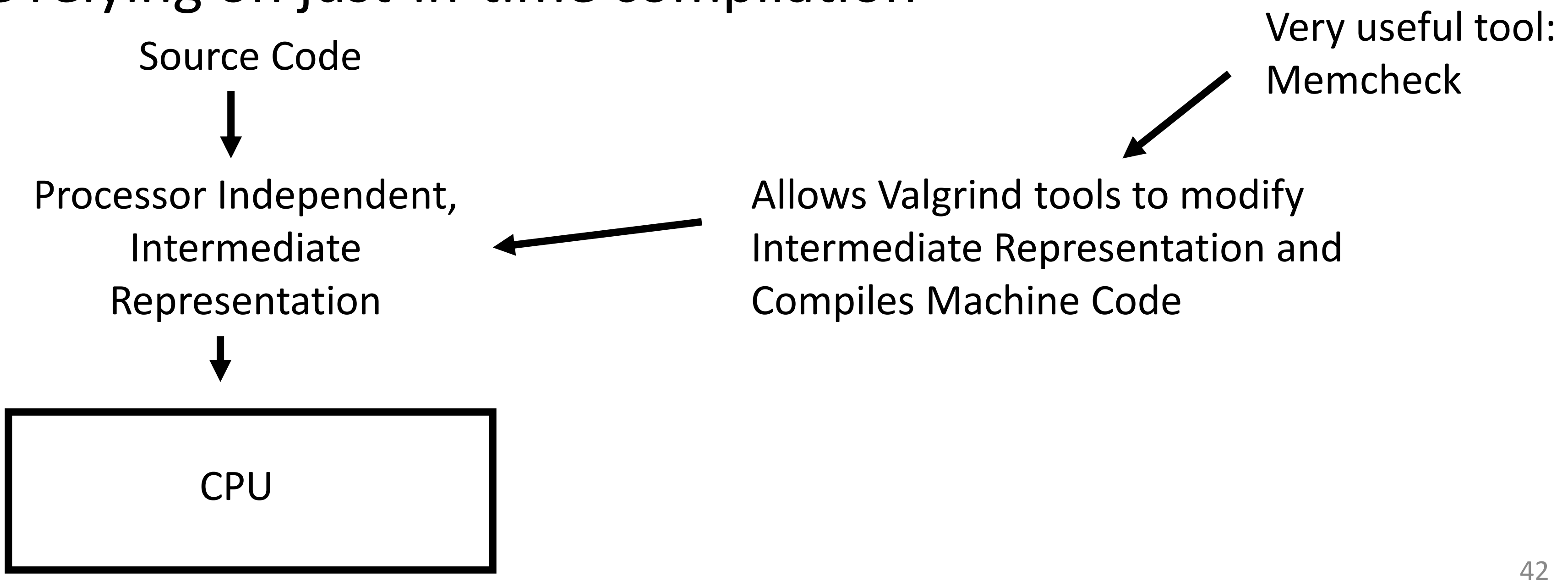
- Varying degrees of „quality“ of pattern checking (with varying Speed)
  - Only syntactical
    - Do not use function X from library Y
  - More context-sensitive
    - Number of allocations and free operations should balance
- Policies can be defined by company
- Policies can restrict to use of subset of language only
  - MISRA-C

Binary Code Instrumentation

- Adds additional information to binary code
  - Resulting binary much larger than stripped-off binary
  - Typically used for debugging
  - Allows to connect execution of binary to source code
  - Not shipped in practice to users as it may greatly help attacker, e.g. when trying to decompile (generate code from binary in highlevel language with equivalent functionality)
- General techniques for debugging (non-specific to security analysis)
  - Step-wise execution
  - Breakpoints, i.e. program execution halts at specific points
  - Watchpoints , i.e. if certain conditions are met, the debugger gives feedback and may halt the execution

Binary Code Instrumentation

- Add extra code to resulting binary
- May be used to search for errors that are often security critical
- Example: Valgrind
  - Tool for memory debugging, memory leak detection and profiling
- Virtual machine relying on just-in-time compilation



Valgrinds Memcheck

- Validity
  - All freshly allocated memory is not used until it is initialized first
- Addressability
  - Memory used in new allocations is indeed free
    - In contrast to non-freed memory
- Technically, replaces standard C memory allocator
  - Can keep track of state of memory and implement guards around memory areas to detect buffer overflows
- Memory leaks (allocated memory that is not freed), use-after free, buffer overflows, use of non-initialized memory
- Does not find all security vulnerabilities!
- Costs of Valgrind: program execution about 20-30 times slower
  - Usually, only used in testing phase

Other Concepts

- Check code for compliance with a specific subset of programming language that is expected to deliver better (security-wise) code
  - e.g. MISRA-C

- Advanced concepts: Runtime Monitoring
- Profiling run program and output additional information afterwards