Secure Programming

Dr. Shilpa Chaudhari

Professor

Syllabus

Unit I
Secure programming goals, Taxonomy of vulnerabilities (CWE Top 25), Role of static analysis: capabilities, limitations, Source vs. compiled code analysis, Introduction to tools: SonarQube, Fortify, Semgrep : **Chapter 1 and chapter 2**
Unit II
Code review practices in SDLC, Security code review checklist, OWASP Code Review Guide, Static analysis metrics and reporting, Writing secure coding rules for custom tools
Unit III
Input sanitization and canonicalization, Preventing format string, buffer overflow, integer overflow, Using bounded functions in C (strncpy, snprintf), Runtime protections: Stack canaries, ASLR, DEP
Unit IV
Secure error handling and return code strategies, Exception management best practices, Avoiding resource leaks and dangling pointers, Secure logging principles (avoid PII leakage), Introduction to SEI CERT C Coding Standards
Unit V
Injection (Command, SQL), CSRF, XSS, Open Redirect, Clickjacking, File Inclusion, Session Hijacking, Authentication flaws, Secure cookie practices, SameSite flag, Using OWASP Top 10 and Cheat Sheets, Middleware and third-party risk

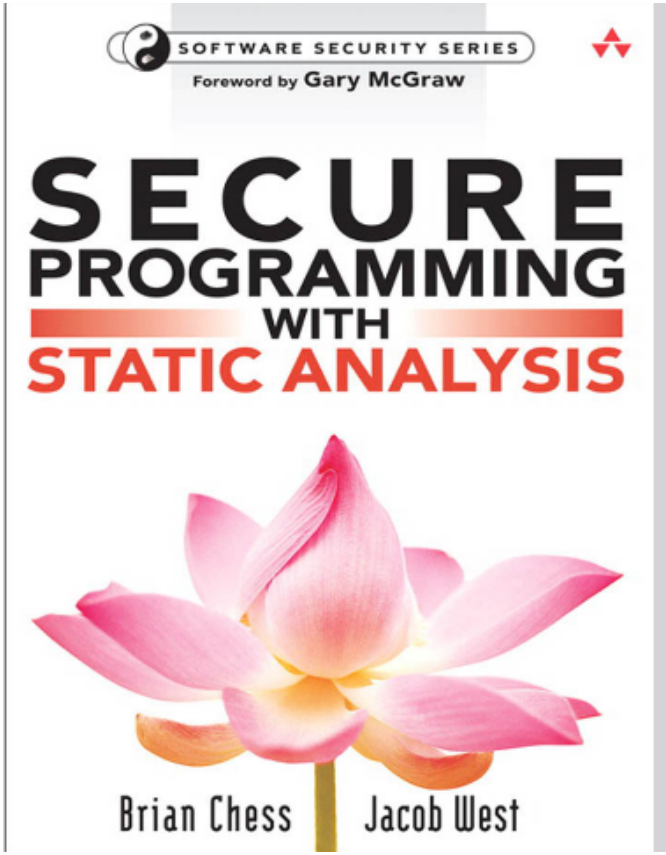
Reference Books

- Chess, B and West, J, Secure Programming with Static Analysis, Addison-Wesley, 2007
- Frank Hissen “Secure Programming of Web Applications”, 2019
- TamaghnaBasu Secure Programming with Python – 31 January 2017
- McGraw, G., Viega, J. (2011). Building Secure Software: How to Avoid Security Problems the Right Way. Addison-Wesley.

Course Outcomes

- At the end of the course students should be able to:
- Identify software security problems solvable via static analysis and secure coding
- Perform security code reviews and interpret metrics from static analysis tools
- Apply secure input handling and memory safety practices in C programs
- Demonstrate effective logging and exception management in secure program design
- Analyze and mitigate common web vulnerabilities using OWASP practices

Secure Programming Lecture 1: Introduction



What is this course about?

- Building software that's more secure
  - … finding flaws in existing software
  - … avoiding flaws in new software (design and code)
  - … techniques, tools and understanding to do this
- The infrastructure around secure software:
  - … language, libraries, run-time; other programs
  - … data storage, distribution, protocols and APIs
  - … development and deployment methods
- And first of all, setting policies for security
  - … what should be protected
  - … who/what is trusted
  - … risk assessment: cost of defences.

Why Security Matters in Software

- High cost of insecure code
- Real-world failures: buffer overflows, injection flaws
- Recent breaches (e.g., SolarWinds, Log4Shell) highlight risks

Safety versus security
Safety is concerned with ensuring bad things don't happen accidently. For example, aeroplanes don't fall out of the sky because maintenance checks are forgotten.

- Security is concerned with with ensuring that bad things don't happen because of malicious actions by others. For example, terrorists cannot drive bombs into airport departure halls.
- The distinction is sometimes blurred, and the two interact in intriguing ways.

Why does security failures happen?

- Ostensibly, many security failures are due to software vulnerabilities. Are they inevitable?
- Many surrounding questions. Can we:
  - … find vulnerabilities (before attacks)?
  - … detect exploits in-the-wild?
  - … repair vulnerabilities (routinely/automatically)?
  - … program better to avoid vulnerabilities?
  - … measure risk associated with software?
  - … design or verify to prevent them?
  - … develop new technology for all the above?
- Questions beyond the technical, too. Can we:
  - … insure against cyber incidents?
  - … regulate for better security?

Common Sources of Vulnerabilities

- Buffer overflows and memory safety issues
- Injection attacks (SQLi, command injection)
- Race conditions and concurrency errors
- CWE Top 25 (2025): Improper input validation, hard-coded secrets

Why Bugs Slip Through

- Developer mindset: focus on features, not threats
- Time pressure and increasing software complexity
- Traditional testing misses corner cases

- Solution is to use a secure programming approach - the practice of designing, writing and maintaining code of software in a way that prevents security vulnerabilities by incorporating security into the coding phase by using secure coding practices.

Programming vs. Secure Programming

- Programming ensures functional correctness
- Secure programming ensures resilience against attacks
- Example: input validation vs. unchecked input

Challenges of Secure Programming

Malicious Influence

- Injection Attacks (e.g., SQL injection, cross-site scripting (XSS), and denial-of-service (DoS))
- Phishing and Social Engineering (e.g. Email Phishing , Impersonation)
- Supply Chain Attacks (hacking third-party libraries)

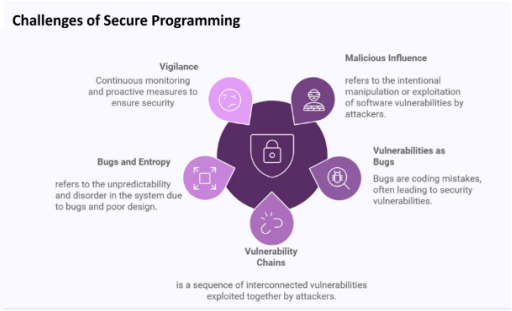
The Impact of such attacks Compromise system **integrity** and **Confidently** (Unauthorized data access or manipulation)

Vulnerability Chains

- An attacker might first exploit a buffer overflow vulnerability to execute code, and then exploit a privilege vulnerability to gain administrative access.

Vigilance - Elements of Vigilance:

- Regular Code Audits
- Automated Security Tools (static/dynamic analysis)
- Security Awareness Training for developers
- Incident Response Planning



Vulnerabilities as Bugs

- Buffer Overflows
- Race Conditions
- Improper Input Validation

The Impact of such bugs lead to security breaches and Unintended system behavior.

Bugs and Entropy

Ex: Deserialization , Buffer Overflows, Race Conditions Complexity in code and Unmanaged legacy code lead to high entropy.

Consequences of High Entropy:

- Increased number of bugs
- Unpredictable system behavior
- Difficult maintenance

The challenge of software security

- Software artefacts are among the most complex built.
  - … Design flaws are likely
  - … Bugs seem inevitable
- Flaws and bugs lead to vulnerabilities which are exploited by attackers.
- Often to learn secrets, obtain money. But many other reasons: a security risk assessment for a system should consider different attackers and their motives.

Secure Programming Goals

- Go Beyond Functional Correctness – Ensure safety under malicious inputs
- Anticipate and Defend Against Attackers – Think like an adversary
- Eliminate Common Vulnerabilities Early – Prevent buffer overflows, injections, race conditions
- Build Resilience – Defense in depth and safe degradation
- Make Security Measurable and Repeatable – Static analysis, coding standards
- Balance Security with Usability and Performance

Software development methodologies are used to achieve these goals

Software development methodologies

- Most software development methodologies can be cast into some arrangement of the same four steps:

1. Plan—Gather requirements, create a design, and plan testing.

2. Build—Write the code and the tests.

3. Test—Run tests, record results, and determine the quality of the code.

4. Field—Deploy the software, monitor its performance, and maintain it as necessary.

The only way to get security right is to incorporate security considerations into all the steps: late-in-the-game approach

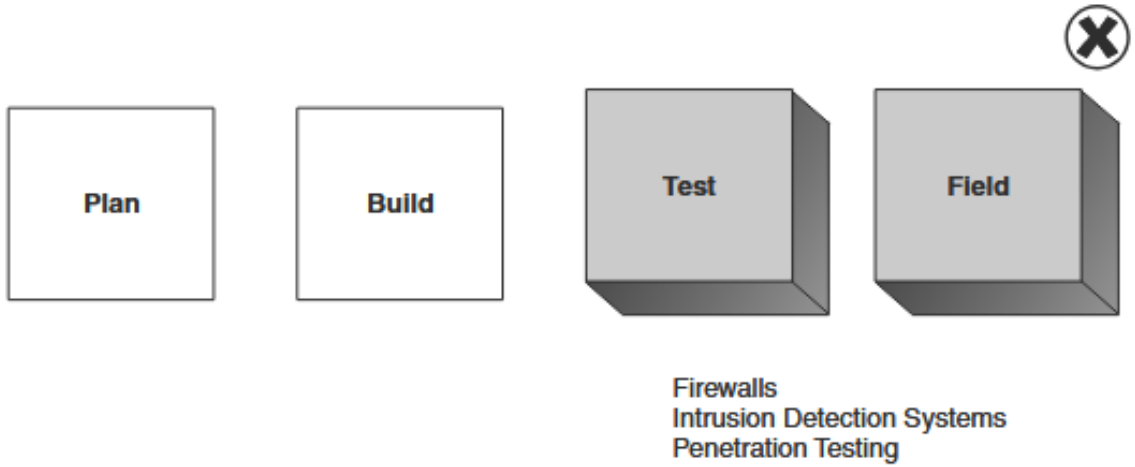


Figure 1.2 Treating the symptom: Focusing on security after the software is built is the wrong thing to do.

- it doesn't work

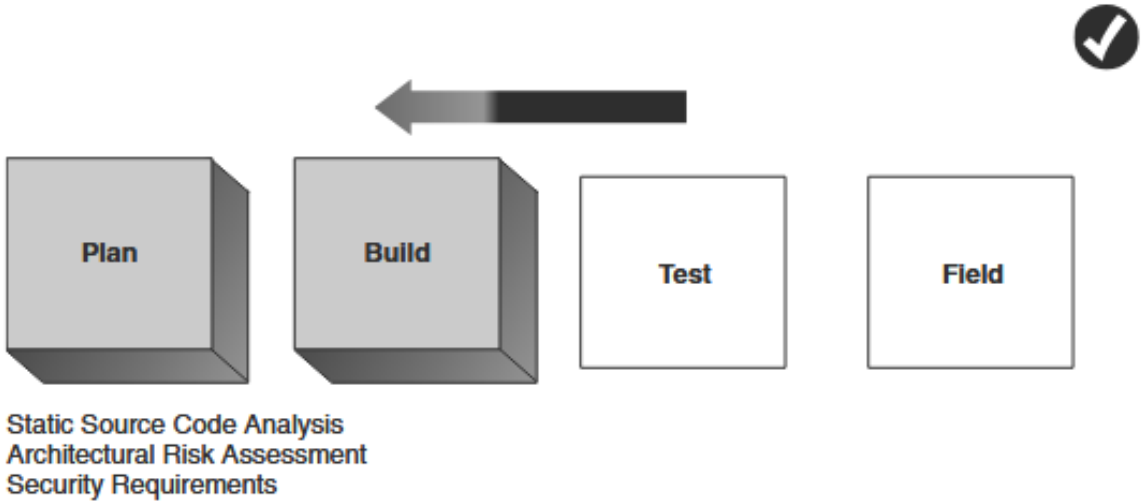


Figure 1.3 Treating the cause: Focusing on security early, with activities centered on the way the software is built.

- Security is an integral part of the way software is planned and built - McGraw lays out a set of seven touchpoints for integrating software security into software development

Taxonomy of vulnerabilities

- It is impossible to study vulnerabilities for very long without beginning to pick out patterns and relationships between the different types of mistakes that programmers make.
- From a high level, defects are divided into two loose groups: generic and context-specific
  - A generic defect is a problem that can occur in almost any program written in the given language. Eg. buffer overflow
  - Finding context-specific defects requires a specific knowledge about the semantics of the program at hand. Eg. Program that handles credit card numbers.

Taxonomy of vulnerabilities

- The best way to find a particular defect depends on whether it is generic or context specific, and whether it is visible in the code or only in the design

- Common Weakness Enumeration (CWE) project (http://cve.mitre.org/cwe/) is building a formal list and a classification scheme for software weaknesses.
- The OWASP Honeycomb project (http://www.owasp.org/index.php/Category:OWASP_Honeycomb_Project) is using a community-based approach to define terms and relationships between security principles, threats, attacks, vulnerabilities, and countermeasures.

Taxonomy of vulnerabilities

The Seven Pernicious Kingdoms are : this classification works well for describing both generic defects and context-specific defects.

1. **Input Validation and Representation -** Flaws in how input is checked, filtered, or encoded (e.g., SQL injection, XSS).
2. **API Abuse -** Misuse or misunderstanding of APIs, leading to insecure states (e.g., incorrect crypto API use).
3. **Security Features -** Failures in implementing authentication, access control, encryption, auditing, etc.
4. **Time and State -** Issues with race conditions, ordering, synchronization, or multi-threading.
5. **Error Handling -** Poor exception or error management that leaks information or bypasses checks.
6. **Code Quality -** Bugs like buffer overflows, memory leaks, uninitialized variables.
7. **Encapsulation -** Breaking boundaries between components or violating abstraction principles.

* Environment

Taxonomy of vulnerabilities

- The Seven Pernicious Kingdoms in relation to the OWASP Top 10

Seven Pernicious Kingdoms	OWASP Top 10
1. Input Validation and Representation	1. Unvalidated Input
	4. Cross-Site Scripting (XSS) Flaws
	5. Buffer Overflows
	6. Injection Flaws
2. API Abuse	
3. Security Features	2. Broken Access Control
	3. Broken Authentication and Session Management
	8. Insecure Storage
4. Time and State	
5. Error Handling	7. Improper Error Handling
6. Code Quality	9. Denial of Service
7. Encapsulation	
* Environment	10. Insecure Configuration Management

Taxonomy of Vulnerabilities (CWE Top 25, 2025)

- Out-of-bounds write/read
- Improper input validation
- Use after free, double free
- Hard-coded credentials
- Cross-site scripting (XSS)

Software security touchpoints

- Putting software security into practice requires making some changes to the way most organizations build software.
  - Security can be interleaved into existing development processes
- How to work security engineering into existing development processes - requirements, architecture, design, coding, testing, validation, measurement, and maintenance
- The secure software development process “touchpoints”, in priority order:

1. Code review and repair
2. Architectural risk analysis
3. Penetration testing
4. Risk-based security testing
5. Abuse cases
6. Security requirements
7. Security operations

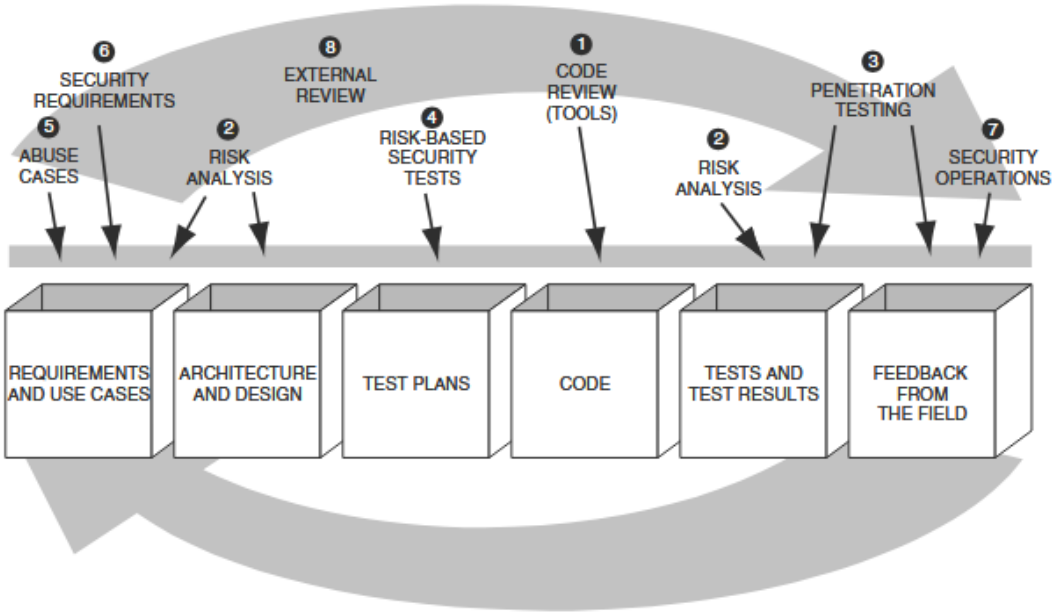
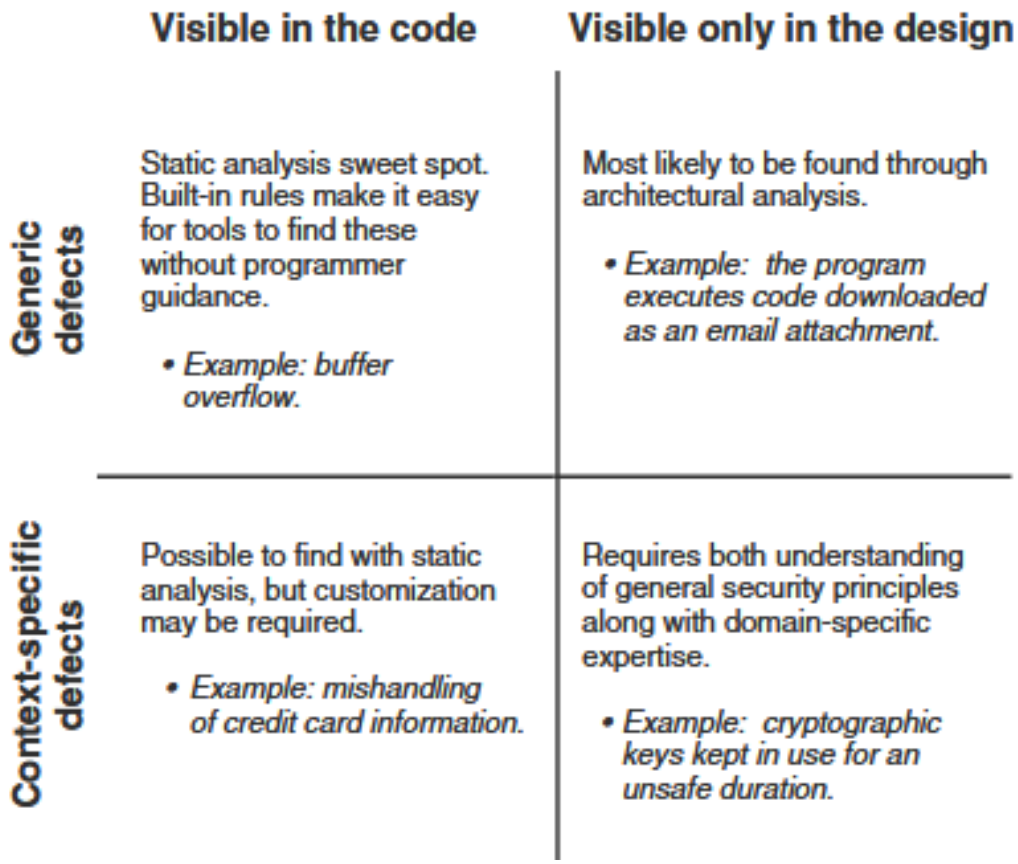


Figure 2 The software security touchpoints as introduced and fleshed out in *Software Security: Building Security In.*

Software security touchpoints

- Static analysis techniques help with code review and repair. Some advanced static analysis techniques may help with architectural (design) understanding too.
- Design flaws are best found through architectural analysis. They may be generic or context-specific.
  - Generic flaws - Bad behaviour that any system may have. e.g., revealing sensitive information
  - Context-specific flaws - Particular to security requirements of system. e.g., key length too short for long term usage
- Security programming bugs (sometimes more serious flaws) are best found through static code analysis.
  - Generic defects- Independent of what the code does; May occur in any program; May be language specific e.g., buffer overflow in C or C++
  - Context-specific defects - Depend on particular meaning of the code; Even when requirements may be general; Language agnostic. AKA logic errors.e.g., PCI DSS rules for CC number display violated

Vulnerabilities matrix



Common Weakness Enumeration

- Weaknesses classify Vulnerabilities
- A CWE is an identifier such as CWE-287
- CWEs are organised into a hierarchy
- The hierarchy (perhaps confusingly) allows:
  - multiple appearances of same CWE
  - different types of links
  - different ways of grouping CWEs
- This allows multiple views
  - different ways to structure the same things
  - also given CWE numbers
- E.g., the Top 2024 CWE Top 25 is CWE-787.
- See https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html
- Exercise. Browse the CWE hierarchy to understand representative weaknesses in each category.

1430 - Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses

- Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - *(79)*
- Out-of-bounds Write - *(787)*
- Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - *(89)*
- Cross-Site Request Forgery (CSRF) - *(352)*
- Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') - *(22)*
- Out-of-bounds Read - *(125)*
- Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - *(78)*
- Use After Free - *(416)*
- Missing Authorization - *(862)*
- Unrestricted Upload of File with Dangerous Type - *(434)*
- Improper Control of Generation of Code ('Code Injection') - *(94)*
- Improper Input Validation - *(20)*
- Improper Neutralization of Special Elements used in a Command ('Command Injection') - *(77)*
- Improper Authentication - *(287)*
- Improper Privilege Management - *(269)*
- Deserialization of Untrusted Data - *(502)*
- Exposure of Sensitive Information to an Unauthorized Actor - *(200)*
- Incorrect Authorization - *(863)*
- Server-Side Request Forgery (SSRF) - *(918)*
- Improper Restriction of Operations within the Bounds of a Memory Buffer - *(119)*
- NULL Pointer Dereference - *(476)*
- Use of Hard-coded Credentials - *(798)*
- Integer Overflow or Wraparound - *(190)*
- Uncontrolled Resource Consumption - *(400)*
- Missing Authentication for Critical Function - *(306)*

Role of Static Analysis

- Definition: Analyzes source/compiled code without execution
- Static analysis for security - A perfect fit for security because in principle:
  - it examines every code path, and
  - it considers every possible input
- Only a single path/input is needed for a security breach.
- Dynamic testing only reaches paths determined by test cases and only
- uses input data given in test suites.
- Other advantages:
  - often finds root cause of a problem
  - can run before code complete, even as-you-type
- But also some disadvantages/challenges.

Capabilities of Static Analysis

- Detects vulnerabilities automatically
- Helps enforce secure coding guidelines
- Integration with CI/CD pipelines for DevSecOps

Limitations of Static Analysis

- False positives and negatives are common
- Scalability and performance issues
- Requires developer adoption and tuning
- Perfect static security analysis is impossible for solving an impossible task

False positive: false alarms on secure programs

- Because the security or correctness question must be approximated, tools cannot be perfectly precise. They may raise false alarms, or may miss genuine vulnerabilities.
  - a problem reported in a program when no problem actually exists
- The false positive problem is hated by users:
  - too many potential problems raised by tool
  - programmers have to wade through long lists
  - true defects may be lost, buried in details
- Modern tools minimise false positive rates for usability.

False negatives: defective programs not caught

- In practice, tools trade-off false positives with false negatives, missing defects.
  - a problem exists in the program, but the tool does not report it.
- Risky for security:
  - one missed bug enough for an attacker to get in!
- Academic research usually concentrates on sound techniques (an algorithm guarantees some security property), with no false negatives.
- But strong assumptions are needed for soundness. In practice, tools must accept missing defects.
- How are imprecise tools measured and compared? It is difficult. The US NIST SAMATE project has worked on static analysis benchmarks.

Static analysis in practice

- Correctness is undecidable in general
  - focus on decidable (approximate) solution
  - or semi-decidable + manual assistance/timeouts
- Avoiding state-space explosion exploring all paths
  - must design/derive abstractions
  - data: restricted domains (abstract interpretation)
  - code: approximate calling contexts
- Environment is unknown
  - program takes input (maybe even code) from outside
  - other factors, e.g., scheduling of multiple threads
  - again, use abstractions and simplified assumptions
- Complex behaviours difficult to specify
  - use generic specifications (overflow, NPE)
  - so-called lightweight methods

Static analysis jobs

- A range of jobs can be undertaken by static analysis or ensuring secure code:
  - Basic task
    - Style checking: ensuring good practice
    - Type checking: maybe as part of language
  - More advanced tasks
    - Program understanding: inferring meaning
    - Property checking: ensuring no bad behaviour
    - Program verification: ensuring correct behaviour
    - Bug finding: detecting likely errors
- General tools in each category may be useful for security. Dedicated static security analysis tools also exist. Examples are Fortify and Coverity, both now integrated into larger secure development products.
- Other popular tools include Snyk and CodeQL (available standalone or integrated into GitHub CI).

Style checking for good practice

- Informally, comparing with natural language (intuition)
  - type system: becomes part of syntax of language
  - style checking: a bit like grammar checking in natural language
- Style checking traditionally covers good practice
  - syntactic coding standards (layout, bracketing etc)
  - naming conventions (e.g., UPPERCASE constants)
  - lint-like checking for dubious /non-portable code
    - Modern languages are stricter than old C (or have fewer implementations)
    - style checking becoming part of compiler/IDE
    - but also dedicated tools with 1,000s rules
- Example tools: PMD, Parasoft

Style checking for good practice

```
typedef enum { RED, AMBER, GREEN } TrafficLight;

void showWarning(TrafficLight c)
{
  switch (c) {
  case RED:
    printf("Stop!");
  case AMBER:
    printf("Stop soon!");
  }
}
```

Style as safe practice

- Legal in C language, type checks and compiles fine:

  [dice] da: gcc enum.c

- But with warnings:

```
[dice] da: gcc -Wall enum.c
enum.c: In function ‘showWarning’:
enum.c:7:3: warning: enumeration value ‘GREEN’ not handled in switch [-Wswitch]
switch (c) {
^
```

Type systems: a discipline for programming

- Proper type systems provide strong guarantees
  - Java, ML, Haskell: no type/memory corruption
  - These are strongly typed languages
- Sometimes frustrating: seen as a hurdle. Old joke:
  - When your Haskell program finally type-checks, it must be right!
- Do programmers accept type systems?
  - yes: type errors are necessary, not “false”
  - no: they’re overly restrictive, complicated
    - . . . likely influence on rise of scripting languages
- Nowadays: types are important for reliability, security
  - idea of gradual typing
  - “subset” languages Hack (from PHP) and TypeScript (from JavaScript)

False positives in type checking

```
short s = 0;
int i = s;
short r = i;
```

```
[dice]da: javac ShortLong.java
ShortLong.java:5: error: possible loss of precision
    short r = i;
              ^
  required: short
  found:    int
1 error
```

```
int i;
if (3 > 4) {
        i = i + "hello";
}
```

```
[dice]da: javac StringInt.java
StringInt.java:5: error: incompatible types
        i = i + "hello";
            ^
  required: int
  found:    String
```

No false positives in Python

```
i = 0;
if (4 < 3):
i = i + "hello";
```

- The other way around gives an error in execution:

```
Traceback (most recent call last):
  File "src/stringint.py", line 3, in <module>
    i = i + "hello";
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type systems: intrinsic part of the language

- In a statically type language, programs that can't be type-checked don't even have a meaning.
  - Compiler will not produce code
  - So code for ill-typed programs cannot be executed
  - Programming language specifications (formal semantics or plain English): may give no meaning, or a special meaning

Type systems: flexible part of the language

- In practice, programmers and IDEs do give meaning (sometimes even execute) partially typed programs.
- Recent research: gradual typing (and related work) to make this more precise:
  - start with untyped scripting language
  - infer types in parts of code where possible
  - manually add type annotations elsewhere
  - . . . so compiler recovers safety in some form
  - A good example is TypeScript
- Sometimes even strongly-typed languages have escape routes, e.g., via C-library calls or abominations like unsafePerformIO

Type systems: motivating new languages

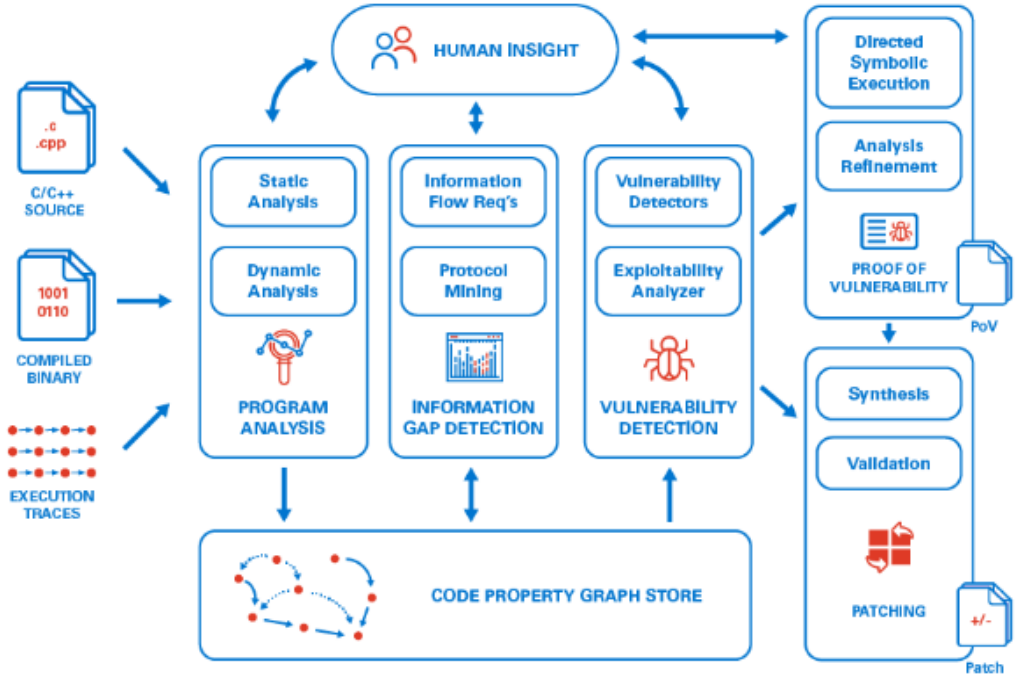
- High-level languages arrived with strong type systems early on (inspired from mathematical ideas in functional languages, e.g., Standard ML, Haskell).
- Language designers asked if static typing can be provided for systems programming languages, without impacting performance too much. Two prominent young examples:
  - Go (2007-) supported by Google
  - Rust (2009-) supported by Mozilla
- both are conceived as type safe low-level languages with built-in concurrency support.

Type systems: modularity advantage

- By design, types provide modularity.
  - write programs in separate pieces
  - type check the pieces
  - put the types together: the whole is type-checked
- This property extends to the basic parts of the language: we find the type of an expression from the type of its parts. Programming language researchers call this compositionality.
- Because of this type systems are a good way to define new static analyses for particular purposes.
- Unfortunately security is often a non-compositional property.

Galois MATE

- A tool made available as open source in 2022 is Galois Inc's MATE, "Merged Analysis To prevent Exploits".

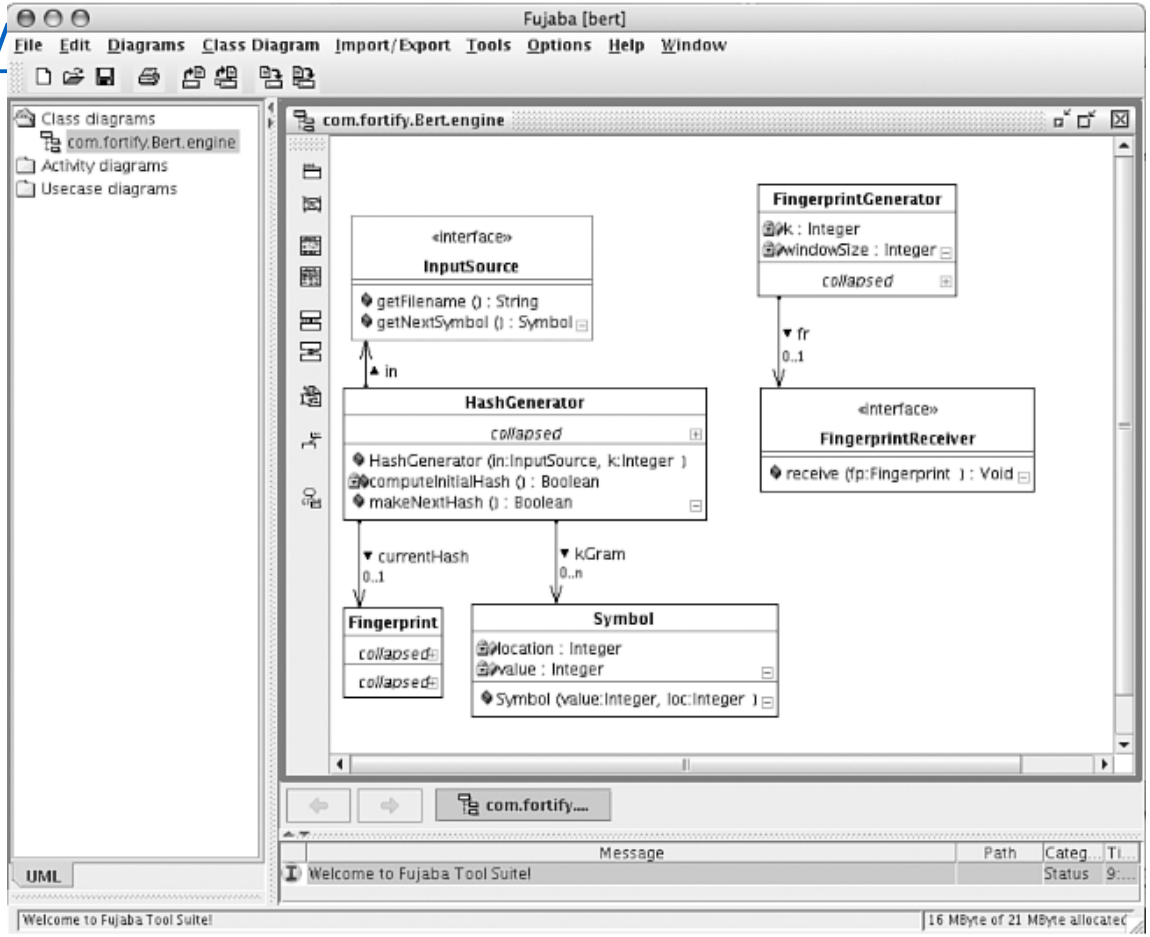


Program understanding tools

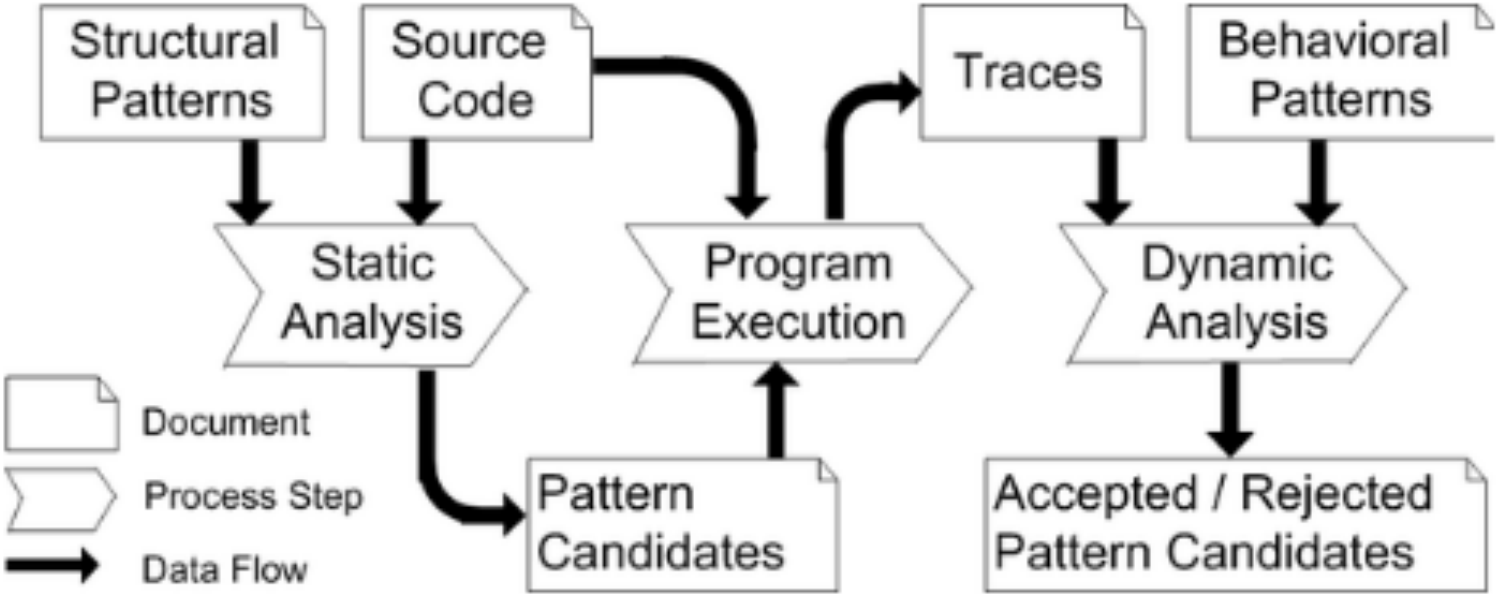
- Help developers understand and manipulate large codebases.
  - Navigation swiftly inside the code
    - finding definition of a constant
    - finding call graph for a method
  - Support refactoring operations
    - re-naming functions or constants
    - move functions from one module to another
    - needs internal model of whole code base
  - Inferring design from code
  - Reverse engineer or check informal design

Program understanding tools – Fujaba, CAST

- The open source Fujaba tool suite (http://wwwcs.uni-paderborn.de/cs/fujaba/) enables a developer to move back and forth between UML diagrams and Java source code.
  - Fujaba can also infer design patterns from the source code it reads.
- CAST Systems (http://www.castsoftware.com) focuses on cataloging and exploring large software systems



Program understanding tools - How Reclipse works



- https://web.cs.upb.de/archive/fujaba/

Program verification

A program verification tool accepts a specification and a body of code and then attempts to prove that the code is a faithful implementation of the specification.

- Uses formal methods
  - e.g., theorem proving and model checking
- Drawback: needs precise formal specification
- Drawback: expensive to industry
  - time consuming, needs experts in logic and maths
- Currently mainly used in safety critical domains
  - e.g., railway, nuclear, aeronautics
  - emerging: automobile, sometimes security

Program verification - equivalence checking

- If the specification is a complete description of everything the program should do, the program verification tool can perform equivalence checking to make sure that the code and the specification exactly match
- Historically, equivalence checking tools struggled with large programs
- Rare in practice due to difficulty of writing complete specifications.
- Example: Orange Book (1980s attempt at mainstream adoption).\
- Property Checking - More common than equivalence checking

Property checking

- Uses partial specifications describing only parts of program behavior.
- Lightweight formal methods
  - Make specifications be standard and generic
    - this program cannot raise NullPointerException
    - all database connections are closed after use
- Static checking
  - Prevent many violations of specification, not all
  - Preconditions (requires) & postconditions (ensures)
  - May produce counterexamples to explain violations
- Approaches: Logical inference, Model checking.

Property checking-Temporal Safety Properties

- Specify sequences of events that must NOT occur.
- Example: A memory location must not be read after being freed.
- Programmers can define custom specifications.
- C code Example snippet: Memory Leak
  1. inBuf = (char*) malloc(bufSz);
  2. if (inBuf == NULL) return -1;
  3. outBuf = (char*) malloc(bufSz);
  4. if (outBuf == NULL) return -1;
- Issue: If first malloc() succeeds and second fails, memory is leaked.
- Violation of property: 'allocated memory should always be freed'
- line 2: inBuf != NULL
- line 5: outBuf == NULL
- line 6: function returns (-1) without freeing inBuf

Soundness of Property Checking Tools

- Sound tool: always reports a problem if one exists (no false negatives).
- Constraints: may disallow recursion, function pointers, or pointer aliasing.
- Tool fails to recognize malloc() returning NULL means no allocation.
- False positives may arise from specification errors or tool limitations.

```
Violation of property "allocated memory should always be freed":
  line 2: inBuf == NULL
  line 3: function returns (-1) without freeing inBuf
```

- Error: Tool misunderstood malloc() returning NULL → no memory allocated.

Null References: A Billion Dollar Mistake?

- Tony Hoare introduced Null references in ALGOL W back in 1965 “simply because it was so easy to implement”.
- He later called it “my billion-dollar mistake”.
- . . . but he called the C gets function a multi-billion dollar mistake!

Commercial and Research Tools

- Praxis High Integrity Systems: Ada verification tool.
- Escher Technologies: custom language compiled to C++/Java.
- Polyspace and Grammatech: commercial property checking tools.
- Numerous university research projects in verification and property checking.
- Formal verification uses rigorous mathematical approaches.
  - **Orange Book (TCSEC, 1985):** Guided secure system development for U.S. government/military.
    - Called for use of formal methods to verify security properties.
    - Orange Book is now obsolete → replaced by **Common Criteria (ISO/IEC 15408)**.
      - International standard for security evaluation.
      - Provides a framework for specifying and measuring security requirements.
      - Still widely used for **government and military systems**.
      - Builds on foundational concepts from the Orange Book.

Bug Finding

- Purpose: Detect unintended program behaviors, not formatting/style issues.
- Focus: "Bug idioms" – common patterns in code that often indicate bugs.
- Examples:
  - **Double-checked locking** (Java, pre-1.5 → unsafe).
  - Tools like **FindBugs** automatically detect such idioms.

Example – Double-Checked Locking

- The purpose of the code is to allocate at most one object while minimizing the number of times any thread needs to enter the synchronized block
- Problem: Fails in Java 1.5 version and before due to memory model.
- Tool Output (FindBugs): how the open source tool FindBugs (http://www.findbugs.org) identifies the problem

```
1 if (this.fitz == null) {
2   synchronized (this) {
3     if (this.fitz == null) {
4       this.fitz = new Fitzer();
5     }
6   }
7 }
```

```
M M DC: Possible doublecheck on Fizz.fitz in Fizz.getFitz()
        At Fizz.java:[lines 1-3]
```

Characteristics of Bug Finders

- Built-in bug idioms, extendable via inference.
- bug finding tools generally focus on producing a low number of false positives even if that means a higher number of false negatives.
- Ideal tool: **sound w.r.t counterexample** (reported bugs are always feasible).
- Examples:
  - **FindBugs** (Java)
  - **Coverity** (C/C++)
  - **Prefast (/analyze)** in Microsoft Visual Studio
  - **Klocwork**: combines bug finding and program understanding

Security Review Tools

•Early tools: **ITS4, RATS, Flawfinder** → glorified *grep* for risky functions like *strcpy*().
•Limitation: High false positives; best used as **aids in code review**.
•Modern tools: Hybrid of **property checkers + bug finders**.
•Security tools emphasize **minimizing false negatives** → flagging even potential risks
•A C program with two calls to strcpy(). A good security tool will categorize the first call as safe (though perhaps undesirable) and the second call as dangerous.

```
int main(int argc, char* argv[]) {
  char buf1[1024];
  char buf2[1024];
  char* shortString = "a short string";
  strcpy(buf1, shortString); /* safe use of strcpy */
  strcpy(buf2, argv[0]);     /* dangerous use of strcpy */
```

Security Tool Vendors

- **Fortify Software** → comprehensive static analysis for security.
- **Ounce Labs** → security-focused analysis.
- **Secure Software** → acquired by Fortify (2007).
- Security tools best applied **within code review** for accurate interpretation.

Static analysis - core Theoretical Foundations

1. The Halting Problem (Turing, 1936)

- Question: *Does a program halt or run forever?*
- Turing proved this is **undecidable** — there is no general algorithm to solve it for all programs.
- Implication: The only guaranteed way to know what a program will do is to actually run it.

2. Rice's Theorem (1953)

- States: *Any non-trivial semantic property of programs is undecidable.*
- Example: "Does this program ever call unsafe()?" → requires solving the halting problem.
- Implication: No static analysis tool can **perfectly** determine interesting program properties for all cases.

Practical Consequences for Static Analysis

- **No Perfect Tool Exists**: Every static analyzer must produce either:
  - **False positives** (flagging issues that aren't real), or
  - **False negatives** (missing some real issues).
- **Design trade-off**: Tools balance soundness (no false negatives) vs. completeness (no false positives).
- **Reality check**: Even though "perfect" analysis is impossible, static analysis remains highly useful in practice.

Practical Consequences for Static Analysis Example

- Perfectly determining any nontrivial program property is impossible in the general case.
- is_safe() cannot behave as specified when the function bother() is called on itself.

```
bother(function f) {
  if ( is_safe(f) )
    call unsafe();
}

b = bother;
bother(b);
```

- If is_safe(bother) → true → then bother() calls unsafe() → contradiction.
- If is_safe(bother) → false → then bother() won't call unsafe() → but it's actually safe → contradiction.
- Conclusion: **No function is_safe() can work perfectly.**

Big Picture

- **Undecidability ≠ uselessness**
  → Even though static analysis is *theoretically* limited (halting problem, Rice's theorem), tools are still very valuable.
- Practical Factors that Matter
  - Making Sense of the Program
  - Trade-Offs: Precision, Depth & Scalability
  - Finding the Right Stuff
  - Ease of Use

Making Sense of the Program

- Parsing source code correctly (compilers differ, dialects exist).
- Modeling **libraries and system calls** (thousands of APIs).
- Handling **multi-language systems** and **configurations** (SOA, dependency injection).
- For **web apps**: mapping URIs and parameters to vulnerabilities.
- Static + dynamic tools can complement each other (e.g., generate HTTP exploits, root-cause analysis).

Trade-Offs: Precision, Depth & Scalability

- **Precision**: high precision → expensive (time, memory).
- **Scalability**: large codebases (millions of LOC) require approximations.
- **Depth**: line-level vs. function-level vs. whole program/system.
- **Speed grades**:
  - Instantaneous (IDE spell-check style, e.g., Flawfinder).
  - Coffee break (commit-level check).
  - Overnight/weekend (enterprise-scale, Coverity, Fortify).

Finding the Right Stuff

- Tools must focus on **relevant defects**:
  - Clients vs servers → different priorities.
  - OS vs web app → different failure modes.
- Extensibility → allow **custom rules** (e.g., private data misuse).
- Quality of **ruleset** matters more than raw number of rules.
- **Benchmarking efforts**:
  - SecuriBench, SecuriBench Micro (Livshits).
  - Zitser/Lippman/Leek vulnerable programs.
  - NIST **SAMATE** project.
  - Analyzer Benchmark (ABM).
  - DHS Build Security In samples.
- Benchmarking challenge: No universal yardstick (false positives vs false negatives trade-off).

Ease of Use

- Tools are *bearers of bad news* → must present results **clearly**.
- Error reporting when parsing fails is essential.
- Integration with **build systems and IDEs** (Make, Ant, Maven, Visual Studio, Eclipse).
- Usability features:
  - Suppress false positives (so they don't reappear).
  - Focus only on **new issues** across revisions.
- Better tools allow **user control of false positive vs false negative trade-offs**.

Static analysis tools - analyze source or compiled code

- Static analysis tools can analyze source or compiled code.
- **Source code analysis** examines the program as the compiler sees it, preserving programmer intent, type information, and structure.
- **Compiled code analysis** examines bytecode or executables as the runtime sees them, removing compiler ambiguity but often losing type information and increasing complexity.
- **Advantages of Compiled Code Analysis**
  - Compiler removes ambiguity → no guessing about interpretation.
  - Sometimes **only bytecode/executable is available** (e.g., proprietary software, closed-source libraries).
- **Disadvantages of Compiled Code Analysis**
  - **Decoding binaries is hard** (esp. x86 variable-width instructions).
  - **Loss of type information** → harder to reason about semantics.
  - Compiler optimizations obscure intent.
  - **Mapping results back to source code** is difficult without debug info.

Special Cases

- **Java Bytecode**:
  - Retains type info → easier than native binaries.
  - But compiler transformations obscure programmer's intent (e.g., JSP → 3 lines → 50+ lines of Java code).
- **C/C++ binaries**:
  - Much harder → analysis sees packets, not "SQL query semantics."

- **Source code contains richer semantic info.**
- **Bytecode/executables** may be easier to obtain but harder to analyze meaningfully.
- For **native programs** → analyzing source is clearly better.
- For **Java-like environments** → trade-offs exist.

Summary

- Secure programming is essential for trustworthy software
- Static analysis helps, but not a silver bullet - **highly useful for security**, enabling thorough examination of code that would be infeasible manually.
- All tools produce **false positives and false negatives**:
  - False negatives are more dangerous for security.
  - Excessive false positives can make reviewers ignore real issues.
- Practical challenges for static analysis tools include the following:
  - Making sense of the program (building an accurate program model)
  - Making good trade-offs between precision, depth, and scalability
  - Looking for the right set of defects
  - Presenting easy-to-understand results and errors
  - Integrating easily with the build system and integrated development environments
- **Source vs Compiled Code**
  - **Bytecode languages (Java)** → source and compiled analysis are roughly equal.
  - **C/C++ programs** → source analysis is easier and more effective; compiled code analysis is harder and less precise.

Static analysis tools

Type of Tool/Vendors	Web Site
Style Checking	
PMD	http://pmd.sourceforge.net
Parasoft	http://www.parasoft.com
Program Understanding	
Fujaba	http://wwwcs.uni-paderborn.de/cs/fujaba/
CAST	http://www.castsoftware.com
Program Verification	
Praxis High Integrity Systems	http://www.praxis-his.com
Escher Technologies	http://www.eschertech.com
Property Checking	
Polyspace	http://www.polyspace.com
Grammatech	http://www.gramatech.com
Bug Finding	
FindBugs	http://www.findbugs.org
Coverity	http://www.coverity.com
Visual Studio 2005 \analyze	http://msdn.microsoft.com/vstudio/
Klocwork	http://www.klocwork.com
Security Review	
Fortify Software	http://www.fortify.com
Ounce Labs	http://www.ouncelabs.com