CMPSC 497: Static Analysis

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Our Goal

• In this course, we want to develop techniques to detect vulnerabilities before they are exploited automatically
  ‣ What’s a vulnerability?
  ‣ How to find them?
Static Analysis

- Provides an approximation of behavior
- “Run in the aggregate”
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- “Run in non-standard way”
  - Run in fragments
  - Stitch them together to cover all paths
- Runtime testing is inherently incomplete, but static analysis can cover all paths
Static Analysis

- A challenge is that static analysis is a bit of an art form
  - Which analysis technique do you use to answer which question?
  - That is not so easy
Static Analysis

• A challenge is that static analysis is a bit of an art form

  ‣ Why is it hard?

  ‣ Rice’s Theorem states that all non-trivial questions about the semantic properties of programs from a universal program language are undecidable. (1953)

  • Syntatic properties (e.g., does program have an if-then-else) are possible to answer

  • But, the sort of questions we want to answer are often about semantic properties

  ‣ Thus, static analysis uses approximate program models
Correctness

• How does this impact proving a program is correct?
Correctness

- **Soundness:**
  - Predicted results must apply to *every* system execution
    - Overapproximate the effect of every program statement
  - Absolutely mandatory for trustworthiness of analysis results!

- **Completeness:**
  - Behavior of *every* system execution caught by analysis
  - Prove any true statement in program is really true
    - Usually not guaranteed due to approximation
  - Degree of completeness determines quality of analysis

- **Correctness:** Soundness $\wedge$ Completeness (rare)
Soundness

• **Soundness:**
  ‣ All executions are represented
  ‣ **Implication 1:** no false negatives, as static analysis model represents all executions possible

• However, unlikely that model is a correct representation of the program semantics
  ‣ **Implication 2:** Sound model is not complete
  ‣ **Implication 3:** A sound static analysis will produce some false positives
  ‣ The number of false positives determines the quality of the analysis
Static Analysis Approaches

• A challenge is that static analysis is a bit of an art form
  ‣ Which analysis technique do you use to answer which question?

• How about for control flows, type-based analysis, and taint analysis?
Static Analysis Approaches

• Control flow
  ‣ Does a program execute one statement (e.g., security check) before another statement (e.g., security-sensitive operation)? Ordering of statements

• Type-based analysis
  ‣ Does a program use data lacking properties (e.g., security check) in a statement (e.g., security-sensitive operation)? Label data using types

• Taint analysis
  ‣ Does a program statement use a tainted value, and what is the impact of executing the statement on its variables?
CFG Analysis

• Does your program have a double free?
• Can control flow analysis detect this?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?
• What does the CFG look like?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?
• What is the property of the CFG that indicates violation?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?

• Can we identify the exploitation in this analysis?

```c
foo(int x, char **y) // need not be “main”
{
...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R1);
    free(buf1R2);
}
```
CFG Analysis

- Does your program have a double free?
- What about this code?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf2R1 = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

- Does your program have a double free?
- What about this code? False positive?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf2R1 = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?

• How do we change the property to detect more accurately (with fewer false positives)?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf2R1 = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?
• Does our new rule work for the following?

```c
foo(int x, char **y) // need not be “main”
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    bar(&buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf1R2);
    free(buf1R1);
}
CFG Analysis

• Does your program have a double free?
• What would need to be done to check?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    bar(&buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?
• What about this one?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf3R1 = buf2R1;
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf3R1);
    free(buf1R2);
}
```
Type-based Analysis

• Does your program have a double free?
• Can we express the rule with types (type-based)?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf2R1 = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
    }
```
Type-based Analysis

- Does your program have a double free?
- Can we express the rule with types (type-based)?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    DEF buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1), FREE x1 = buf2R1;
    DEF y = x1, y = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(y), FREE x2 = y;
    free(buf1R2);
}
```
Type-based Analysis

- Does your program have a double free?
- Can we express the rule with types (type-based)?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    DEF buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1), FREE x1 = buf2R1;
    DEF y = x1, y = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(x1); FREE x2 = x1;
    free(buf1R2);
}
```
Type-based Analysis

- Does your program have a double free?
- Can we express the rule with types (type-based)?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    bar(&buf2R1);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf1R2);
}
```
Taint Analysis

• Does your program have a double free?
• How would taint analysis be applied?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1);
    buf2R1 = (char *) malloc(BUFSIZE2);
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf2R1);
    free(buf2R1);
    free(buf1R2);
}
```
Taint Analysis

• Does your program have a double free?
• What is the property to check?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1); // taint
    buf2R1 = (char *) malloc(BUFSIZE2); // untaint
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf1R2); // taint
    free(buf2R1);
    free(buf1R2);
}
```
CFG Analysis

• Does your program have a double free?
• What about this one?

```c
foo(int x, char **y) // need not be "main"
{
    ...
    buf1R1 = (char *) malloc(BUFSIZE2);
    buf2R1 = (char *) malloc(BUFSIZE2);
    free(buf1R1);
    free(buf2R1); // taint
    buf3R1 = buf2R1; // taint
    buf1R2 = (char *) malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1-1);
    free(buf3R1); // taint
    free(buf1R2);
}
```
Analysis Question

- Are these proposed approaches sound?
  - What is the implication of an inaccurate, sound analysis?
Static Analysis Tools

• **Good news** – there are static analysis tools available, so you don’t have to write your own
  ‣ [https://www.owasp.org/index.php/Static_Code_Analysis](https://www.owasp.org/index.php/Static_Code_Analysis)

• Examples of tools (some examples)
  ‣ Fortify (HP)
  ‣ GrammaTech
  ‣ Checkmarx Static Code Analysis
  ‣ Rational AppScan Source Edition (IBM)
  ‣ Coverity
Static Analysis Tools

• **Good news** – there are static analysis tools available, so you don’t have to write your own

• Use of static analysis tools
  

• Description of the usage scenarios and pitfalls of those in practice
Overall Use Process

- Understanding what you are scanning
- Validate the integrity of the results
- Understand how to customize
- Still some manual effort at the end
What Are You Scanning?

- Tools are often language specific
  - Web applications and C/C++/Java are well supported
  - Scripting languages not so much
- Often only scans your executable code — i.e., for what you have source
  - Doesn’t cover libraries
- Amount of manual work can vary
How Do Tools Work?

• Tools convert the program code into an intermediate representation
• Then various analyses are applied to the intermediate representation

• Make sure your code has been translated to the intermediate representation
  ‣ Versioning in language, compilers, any scripts, etc.
How Do Tools Work?

• Tools convert the program code into an intermediate representation

• Then various analyses are applied to the intermediate representation

• A common framework for writing your own analyses is **LLVM**
  ‣ Not backward compatible among versions, so can create errors
Integrity of Scans

• What are you scanning?
  › How good are the rules for that language and type of program?
• Did all of your source code get scanned?
• Were there errors in translation or analysis?
• Worked fine it appears
• Now what?

---

Overall Process

1. Understand What you are Scanning
2. Verify the Integrity of Your Scan
3. Add Custom Rules
4. Add Canaries
5. Understand What Static Analysis Cannot Do

---

Web Application, Library, Trusted App, Mobile Application

Scanning and Verification

Assess results

Add custom rules

Assess results

Add Canaries

Assess results

Address risk not covered by Static Analysis

---

What are you scanning?

- Well Supported Web Application — Using popular frameworks Struts, Spring MVC, etc.
- Languages — Java, C/C++, Ruby
- Not as well supported (your mileage may vary)
- Android Applications
- Smart Card Apps (More Esoteric)
- Trusted Execution Environment (Extremely Esoteric)
- Operating System Components (Kernel, Drivers, etc.)
- Mixed Languages (Java to C/C++ and vice versa)
- Scripting Language (Javascript, PHP, Python, Ruby)
- Niche Languages (Closure, Erlang, Haskell, Scala, etc.)

Difficult to Scan because of Dependencies

Libraries
Customization

• Typically, the default analysis will not be accurate enough

• If sound analysis, what will happen if not accurate?
  ‣ Experiences with Fortify

• Thus, need to apply customization to the analysis rules to achieve goals
  ‣ Iterative process
Customization

- Examples of customizations
  - Sources
    - Of taint
  - Sinks
    - To detect use of tainted data
  - Passthrough
    - To untaint data
    - To retaint data
  - Summaries
    - Of libraries
  - Rules for what is a violation
Customization in Tools

• Customization flexibility in tools

<table>
<thead>
<tr>
<th>Checkmarx</th>
<th>Fortify</th>
<th>Coverity</th>
<th>Veracode</th>
<th>Whitehat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most Flexible</td>
<td>Very Flexible</td>
<td>Flexible</td>
<td>No custom rules allowed</td>
<td>No custom rules allowed</td>
</tr>
</tbody>
</table>

• Fortify

• Coverity

• Checkmarx
  ‣ [https://checkmarx.atlassian.net/wiki/display/KC/Working+with+Queries](https://checkmarx.atlassian.net/wiki/display/KC/Working+with+Queries)

# RSAC

Custom Rules Summary

- Checkmarx
- Fortify
- Coverity
- Veracode
- Whitehat

- Most Flexible
- Very Flexible
- Flexible
- No custom rules allowed
- No custom rules allowed

Why is this important?
Manual Review

• Of results
  ‣ Remove false positives
  ‣ Improve analysis

• Of code for more complex properties
Take Away

• Static analysis techniques are a common way to detect software flaws
  ‣ However, designing your own analyses is an art form

• Demonstrated/reviewed some simple analysis problems

• Fortunately, it is possible to look for vulnerabilities with static analysis tools
  ‣ A variety of tools are available
  ‣ Using them also requires a fair bit of expertise