OPLSS: Static Analysis

(1) Concepts in Static Analysis
(2) Operational / Denotational Semantics
(3) Abstract Interpretation
(4) Automatic Derivation of Static Analysis
OPLSS: Static Analysis

• Reference
  • Xavier Rival and Kwangkeun Yi,
    Introduction to Static Analysis: an Abstract Interpretation Perspective,
    MIT Press, 2020

https://mitpress.mit.edu/9780262043410/
Software Bugs: A Persistent Problem

• Back in the 90’s

- The Patriot Missile (1991)
  - Floating-point roundoff
  - 28 soldiers died

- The Ariane-5 Rocket (1996)
  - Integer Overflow
  - $100M

- NASA’s Mars Climate Orbiter (1999)
  - Meters-Inches Miscalculation
  - $125M
Software Bugs: A Persistent Problem

• Back in the 90’s
  - The Patriot Missile (1991): Floating-point roundoff, 28 soldiers died
  - The Ariane-5 Rocket (1996): Integer Overflow, $100M
  - NASA’s Mars Climate Orbiter (1999): Meters-Inches Miscalculation, $125M

• And now
  - The ‘Heartbleed’ security flaw that affects most of the Internet
  - This dangerous Android security bug could let anyone hack your phone camera
  - What Boeing’s 737 MAX Has to Do With Cars: Software
  - Homeland Security warns that certain heart devices can be hacked
Software Bugs: A Persistent Problem

**COST OF A SOFTWARE BUG**

- If found in Gathering Requirements phase: $100
- If found in QA testing phase: $1,500
- If found in Production: $10,000

- IBM Systems Sciences Institute, 2015
Why Software Still Fails?

Size of Linux Kernel

Avg. Size of Android Apps

GitHub

10M+ New Developers
44M+ New Repositories
87M+ New Pull Requests in 2019
Cost of Software Quality Assurance

“We have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization”
- Bill Gates

Q: What is the solution to improve software quality at low cost?

A: Program analysis
What to Analyze?

CWE Definitions

<table>
<thead>
<tr>
<th>CWE Number</th>
<th>Name</th>
<th>Number Of Related Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>119</td>
<td>Failure to Constrain Operations within the Bounds of a Memory Buffer</td>
<td>12328</td>
</tr>
<tr>
<td>20</td>
<td>Improper Input Validation</td>
<td>7559</td>
</tr>
<tr>
<td>290</td>
<td>Information Exposure</td>
<td>6316</td>
</tr>
<tr>
<td>89</td>
<td>Improper Sanitization of Special Elements used in an SQL Command</td>
<td>5661</td>
</tr>
<tr>
<td>52</td>
<td>Improper Limitation of a PATH Name to a Restricted Directory</td>
<td>2058</td>
</tr>
<tr>
<td>94</td>
<td>Failure to Control Generation of Code</td>
<td>2400</td>
</tr>
<tr>
<td>123</td>
<td>Out-of-Bounds Read</td>
<td>2422</td>
</tr>
<tr>
<td>247</td>
<td>Improper Authentication</td>
<td>1276</td>
</tr>
<tr>
<td>284</td>
<td>Access Control (Authorization) Issues</td>
<td>1672</td>
</tr>
<tr>
<td>618</td>
<td>Use After Free</td>
<td>1269</td>
</tr>
<tr>
<td>130</td>
<td>Integer Overflow or Wraparound</td>
<td>1112</td>
</tr>
<tr>
<td>678</td>
<td>NULL Pointer Dereference</td>
<td>900</td>
</tr>
<tr>
<td>79</td>
<td>Improper Sanitization of Special Elements used in a Command</td>
<td>769</td>
</tr>
<tr>
<td>727</td>
<td>Out-of-Bounds Write</td>
<td>737</td>
</tr>
<tr>
<td>362</td>
<td>Race Condition</td>
<td>615</td>
</tr>
<tr>
<td>618</td>
<td>Improper Link Resolution Before File Access</td>
<td>518</td>
</tr>
<tr>
<td>77</td>
<td>Improper Sanitization of Special Elements used in a Command</td>
<td>489</td>
</tr>
<tr>
<td>460</td>
<td>Uncontrolled Resource Consumption</td>
<td>461</td>
</tr>
<tr>
<td>611</td>
<td>Information Leak Through XML External Entity File Datasource</td>
<td>393</td>
</tr>
<tr>
<td>534</td>
<td>Unauthorized Udpation of File with Dangerous Type</td>
<td>395</td>
</tr>
<tr>
<td>732</td>
<td>Incorrect Permission Assignment for Critical Resource</td>
<td>350</td>
</tr>
<tr>
<td>74</td>
<td>Failure to Randomize Data into a Different Place</td>
<td>367</td>
</tr>
<tr>
<td>738</td>
<td>Use of Hard-coded Credentials</td>
<td>310</td>
</tr>
<tr>
<td>772</td>
<td>Missing Release of Resource after Effective Lifetime</td>
<td>306</td>
</tr>
<tr>
<td>102</td>
<td>Improper Privilege Management</td>
<td>303</td>
</tr>
<tr>
<td>631</td>
<td>URL Redirection to Untrusted Site (Open Redirect)</td>
<td>265</td>
</tr>
<tr>
<td>592</td>
<td>Deserialization of Untrusted Data</td>
<td>257</td>
</tr>
<tr>
<td>124</td>
<td>Unspecified Format String</td>
<td>233</td>
</tr>
<tr>
<td>284</td>
<td>Incorrect Type Conversion or Cast</td>
<td>180</td>
</tr>
<tr>
<td>615</td>
<td>Double Free</td>
<td>173</td>
</tr>
</tbody>
</table>

Heartbleed, 2019
OpenSSL
CVE-2014-0160

Shellshock, 2014
Bash
CVE-2014-6271

goto fail, 2014
MacOS / iOS
CVE-2014-1266
Properties

• Points of interest in programs
  • for verification, bug detection, optimization, understanding, etc

• In this lecture
  • safety properties
  • liveness properties
  • information-flow properties
Safety Property

- A program **never** exhibits a behavior observable within **finite time**
  - “Bad things will never occur”
  - If false, then there exists a **finite counterexample**
- Bad things: integer overflow, buffer overrun, deadlock, etc
- To prove: all executions never reach error states
Safety Property

- A program never exhibit a behavior observable within finite time

- “Bad things will never occur”

- If false, then there exists a finite counterexample

- Bad things: integer overflow, buffer overrun, deadlock, etc

- To prove: all executions never reach error states

(a) Correct executions
(b) An incorrect execution
(c) Proof by invariance
Invariant

- Assertions supposed to be always true

- Starting from a state in the invariant: any computation step also leads to another state in the invariant

- E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”

- Loop invariant: assertion to be true at the beginning of every loop iteration

```java
x = 0;
while (x < 10) {
    x = x + 1;
}
```
Invariant

- Assertions supposed to be always true

- Starting from a state in the invariant: any computation step also leads to another state in the invariant

- E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”

- Loop invariant: assertion to be true at the beginning of every loop iteration

```java
x = 0;
while (x < 10) {
    x = x + 1;
}
```

Loop invariant 1: “x is an integer”
Invariant

- Assertions supposed to be **always true**

- Starting from a state in the invariant: any computation step also leads to another state in the invariant

- E.g., “x has an int value during the execution”, “y is larger than 1 at line 5”

- Loop invariant: assertion to be true at the beginning of every loop iteration

\[
x = 0; \\
\text{while} \ (x < 10) \ { \\
\quad x = x + 1; \\
}\]

Loop invariant 1: “x is an integer”

Loop invariant 2: “0 <= x < 10”
Example: Division-by-Zero

```c
1: int main()
2: {
3:     int x = input();
4:     x = 2 * x - 1;
5:     while (x > 0) {
6:         x = x - 2;
7:     }
8:     assert(x != 0);
9:     return 10 / x;
10: }
```
Example: Division-by-Zero

```c
1: int main(){
2:   int x = input();  // True
3:   x = 2 * x - 1;
4:   while (x > 0) {
5:     x = x - 2;
6:   }
7:   assert(x != 0);
8:   return 10 / x;
9: }
```
Example: Division-by-Zero

```
1: int main(){
2:     int x = input();   // True
3:     x = 2 * x - 1;     // x is an odd number
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```
Example: Division-by-Zero

```c
1: int main(){
2:     int x = input();  // True
3:     x = 2 * x - 1;    // x is an odd number
4:     while (x > 0) {
5:         x = x - 2;
6:     }
7:     assert(x != 0);
8:     return 10 / x;
9: }
```
Example: Division-by-Zero

1: int main(){
2:   int x = input();   // True
3:   x = 2 * x - 1;     // x is an odd number
4:   while (x > 0) {
5:     x = x - 2;       // x is a positive odd number
6:   }
7:   assert(x != 0);    // x is an odd number
8:   return 10 / x;
9: }
Liveness Property

- A program will **never** exhibit a behavior observable only after **infinite time**
- “Good things will eventually occur”
- If false then there exists an **infinite counterexample**
- Good things: termination, fairness, etc
- To prove: all executions eventually reach target states
Liveness Property

- A program will **never** exhibit a behavior observable only after infinite time
  
  - “Good things will eventually occur”
  
  - If false then there exists an **infinite counterexample**

- Good things: termination, fairness, etc

- To prove: all executions eventually reach target states

(a) Correct executions  
(b) An incorrect execution  
(c) Proof by variance
Variant

- A quantity that **evolves towards** the set of target states (so guarantee any execution eventually reach the set)
- Usually, a value that is strictly decreasing for some well-founded order relation
  - Well-founded order: there exists a minimal element
  - E.g.) an expression of integer type that always takes a positive value and strictly decreasing
    
    ```
    x = pos_int();
    while (x > 0) {
        x = x - 1;
    }
    ```
Variant

• A quantity that **evolves towards** the set of target states (so guarantee any execution eventually reach the set)

• Usually, a value that is strictly decreasing for some well-founded order relation

  • Well-founded order: there exists a minimal element

  • E.g.) an expression of integer type that always takes a positive value and strictly decreasing

    ```
    x = pos_int();
    while (x > 0) {
        x = x - 1;
    }
    ```

    *x* is always a positive integer
Variant

- A quantity that **evolves towards** the set of target states (so guarantee any execution eventually reach the set)

- Usually, a value that is strictly decreasing for some well-founded order relation

  - Well-founded order: there exists a minimal element

  - E.g.) an expression of integer type that always takes a positive value and strictly decreasing

    ```
    x = pos_int();
    while (x > 0) {
      x = x - 1;
    }
    ```

    \( x \) is always a positive integer \( \land \) \( x \) is strictly decreasing
Variant

- A quantity that **evolves towards** the set of target states
  (so guarantee any execution eventually reach the set)

- Usually, a value that is strictly decreasing for some well-founded order relation

  - Well-founded order: there exists a minimal element

  - E.g.) an expression of integer type that always takes a positive value and
    strictly decreasing
      
      ```
      x = pos_int();
      while (x > 0) {
        x = x - 1;
      }
      ```

      \[
      \begin{align*}
      x & \text{ is always a positive integer} \quad \land \quad x \text{ is strictly decreasing} \\
      & \Rightarrow \quad \text{The program terminates}
      \end{align*}
      ```
Trace Properties

- A semantic property $P$ that can be defined by a set of execution traces that satisfies $P$

- Safety and liveness properties are trace properties

\[ [P] \subseteq T_{ok} \]

- State properties: defined by a set of states (so, obviously trace properties)

- E.g., division-by-zero, integer overflow

- Any trace property: the conjunction of a safety and a liveness property
Example

- Correctness of a sorting algorithm as trace property

<table>
<thead>
<tr>
<th>Should not fail with a run-time error</th>
<th>Safety or Liveness?</th>
<th>State?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should terminate</td>
<td>Liveness</td>
<td>-</td>
</tr>
<tr>
<td>Should return a sorted array</td>
<td>Safety</td>
<td>O</td>
</tr>
<tr>
<td>Should return an array with the same elements and multiplicity</td>
<td>Safety</td>
<td>X</td>
</tr>
</tbody>
</table>
Information Flow Properties

- Properties stating the absence of dependence between pairs of executions
- Beyond trace properties: so called hyperproperties
- Mostly used for security purposes:
  - e.g.) multiple executions with public data should not derive private data
Information Flow Properties

- Properties stating the absence of dependence between pairs of executions
- Beyond trace properties: so called hyperproperties
- Mostly used for security purposes:
  - e.g.) multiple executions with public data should not derive private data

A pair of executions with insecure information flow

A pair of executions without insecure information flow
Example

- Assume that variables $s$ (secret) and $p$ (public) take only 0 and 1

```
// Program 0
p_out := p_in

// Program 1
p_out := s * p_in

// Program 2
p_out := |rand(p_in) - s|
```
Example

• Assume that variables $s$ (secret) and $p$ (public) take only 0 and 1

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

// Program 0
p_out := p_in

// Program 1
p_out := s * p_in

// Program 2
p_out := |rand(p_in) - s|
Example

- Assume that variables $s$ (secret) and $p$ (public) take only 0 and 1

```
// Program 0
p_out := p_in
```

```
// Program 1
p_out := s \times p_in
```

```
// Program 2
p_out := |rand(p_in) - s|
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Assume that variables $s$ (secret) and $p$ (public) take only 0 and 1

```
// Program 0
p_out := p_in
```
```
// Program 1
p_out := s * p_in
```
```
// Program 2
p_out := |rand(p_in) - s|
```

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$s$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Theorem (Rice’s theorem). Any non-trivial semantic properties are undecidable.

Undecidable
⇒ Automatic, terminating, and exact reasoning is impossible
Undecidable

⇒ *Automatic, terminating*, and *exact* reasoning is impossible
⇒ If we give up one of them, it is *computable!*
Toward Computability

Undecidable

⇒ **Automatic, terminating, and exact** reasoning is impossible
⇒ If we give up one of them, it is **computable**!

- **Manual** rather than **automatic**: assisted proving
  - require expertise and manual effort
- **Possibly nonterminating** rather than **terminating**: testing, model checking
  - require stopping mechanisms such as timeout
- **Approximate** rather than **exact**: static analysis
  - report spurious results
Soundness and Completeness

• Given a semantic property $\mathcal{P}$, and an analysis tool $A$

• If $A$ were perfectly accurate,

  For all program $p$, $A(p) = \text{true} \iff p \text{ satisfies } \mathcal{P}$

  which consists of

  For all program $p$, $A(p) = \text{true} \implies p \text{ satisfies } \mathcal{P}$ \hspace{1cm} (soundness)

  For all program $p$, $A(p) = \text{true} \impliedby p \text{ satisfies } \mathcal{P}$ \hspace{1cm} (completeness)
Soundness and Completeness

(a) Programs

(b) Sound, incomplete analysis

(c) Unsound, complete analysis

(d) Legend

- Programs that satisfy $\mathcal{P}$
- Programs that do not satisfy $\mathcal{P}$
- Programs for which the analysis returns true
- Programs for which the analysis returns false
Assisted Proving

- Machine-assisted proof techniques
- Relying on user-provide invariants
- Using proof assistants (e.g., Coq, Isabelle/HOL)

- Sound and complete (up to the ability of the proof assistant)
- require manual effort / expertise

- Example: CompCert (verified C compiler), seL4 (verified microkernel)
Testing

• Check a set of finite executions
  • e.g., random testing, concolic (concrete + symbolic) testing

• In general, unsound yet complete
  • Unsound: cannot prove the absence of errors
  • Complete: produce counterexamples (i.e., erroneous inputs)

• Further reading:
Model Checking

- Automatic technique to verify if a model satisfies a specification
  - Model of the target program (finite automata)
  - Specification written in logical formula
  - Verification via exhaustive search of the state space (graph reachability)
- **Sound and complete with respect to the model**
  - May incur infinite model refinement steps
- Example: SLAM (MS Windows device driver verifier)
Model Checking Overview
Model

- Finite state machines constructed manually or by some automatic tools
- Gap between models (finite systems) and programs (infinite systems)
  - either unsound or incomplete with respect to the target program
- Techniques to automatically refine the model on demand
  - may continue indefinitely so stopping mechanisms are required
Example: Double Locking

Calls to lock and unlock must alternate
Example: Drop Root Privilege

"User applications must not run with root privilege"

When exec is called, must have suid ≠ 0

*Hao Chen, David Wagner, and Drew Dean. Setuid Demystified, USENIX Security Symposium, 2002
Specification

- Written in a formal language: modal logic
  - Modal logic = propositional logic + \{necessarily, possibly\}
  - Esp., truth values of assertions vary with time (temporal logic)
  - E.g., LTL (linear temporal logic), CTL (computational tree logic)
- Describe assertions on program properties
  - “x is always positive”, “x can be positive”, “x remains positive until y is negative”, “x is positive after state s”, …
Example: Model & Specification

Target Program

```c
int main(){
1:   int x = input();
2:   x = 2 * x - 1;
3:   while (x > 0) {
4:     x = x - 2;
5:   }
6:   assert(x != 0);
7:   return 10 / x;
}
```
Example: Model & Specification

- State = Label × {Even, Odd, Zero, Error} : finite
- Specification: “The error state is unreachable from the initial states”
- Initial states: {<1, Even>, <1, Odd>}

```c
int main()
{
    int x = input();
    x = 2 * x - 1;
    while (x > 0) {
        x = x - 2;
    }
    assert(x != 0);
    return 10 / x;
}
```
Example: Model & Specification

- State = Label × {Even, Odd, Zero, Error} : finite

- Specification: “The error state is unreachable from the initial states”

- Initial states: {<1, Even>, <1, Odd>}

```c
int main(){
    int x = input();
    x = 2 * x - 1;
    while (x > 0) {
        x = x - 2;
    }
    assert(x != 0);
    return 10 / x;
}
```
Example: Reachability Check

- Check the reachability of the error state from the initial states
- Unreachable: verified
- Reachable and counter example: real bug or spurious warning (why?)

```c
int main()
{
    int x = input();
    x = 2 * x - 1;
    while (x > 0) {
        x = x - 2;
    }
    assert(x != 0);  
    return 10 / x;
}
```
Spurious Reachability

(Finite) Model is an abstraction of the (infinite) target program
Abstraction Refinement

- Automatically refine the model when a spurious counterexample is found
  - New model: to conclude the spurious error is infeasible
  - Until a real counterexample is found or a proof is completed
- May not terminate
Iterative Abstraction Refinement

- CEGAR: CounterExample-Guided Abstraction Refinement

![Diagram of iterative abstraction refinement process]
Summary of Model Checking

- Model (FSM) + Specification (Modal logic) + Verification (Reachability check)

- Theoretical characteristics:
  - If a model checker says “Yes”, the property is guaranteed to hold (Sound)
  - If a model checker says “No”,
    - the counterexample is either a real bug or a spurious warning
  - (refinement; verification)$^+$ until “Yes”, a real bug found, or timeout

- Further reading:
  Model Checking, E. M. Clarke, O. Grumberg, D. Kroening, D. Peled and H. Veith, 2018
Static Analysis

- **Over-approximate** (not exact) the set of all program behavior
- In general, **sound and automatic, but incomplete**
  - May have spurious results
- Based on a foundational theory: Abstract interpretation
- Variants:
  - under-approximating static analysis: automatic, complete, unsound
  - bug finder: automatic, unsound, incomplete, and heuristics
- Example: type systems, ASTREE, Facebook Infer, Sparrow, etc
Industrial Applications of \textit{Astrée}

The main applications of \textit{Astrée} appeared two years after starting the project. Since then, \textit{Astrée} has achieved the following unprecedented results on the static analysis of synchronous, time-triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language:

- In \textbf{Nov. 2003}, \textit{Astrée} was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C analyzed in 1\textsuperscript{st}20 on a 2.8 GHz 32-bit PC using 300 Mb of memory (and 50mn on a 64-bit AMD Athlon™ 64 using 580 Mb of memory).

- From \textbf{Jan. 2004} on, \textit{Astrée} was extended to analyze the electric flight control codes then in development and test for the A380 series. The operational application by Airbus France at the end of 2004 was just in time before the A380 maiden flight on Wednesday, 27 April, 2005.

- In \textbf{April 2008}, \textit{Astrée} was able to prove completely automatically the absence of any RTE in a C version of the automatic docking software of the Jules Vernes Automated Transfer Vehicle (ATV) enabling ESA to transport payloads to the International Space Station [32].
Approximation

- Compute approximated (inaccurate) semantics instead of exact semantics
  - Inaccurate ≠ incorrect
  - E.g., reality: \{2, 4, 6, 8, \ldots\}
    - answer 1: “even” (exact)
    - answer 2: “positive” (conservative)
    - answer 3: “multiple of 4” (omissive)
    - answer 4: “odd” (wrong)
- Given a program and property, the analysis may answer “Yes”, “No”, or “Don’t know” because of approximation
- Key point: choosing a right approximation to prove a given target property
Principle of Static Analysis

- How to design a sound approximation of real executions?
- How to guarantee the termination of static analysis?

A: Abstract Interpretation
Summary

- Different techniques for program reasoning due to the **computability barrier**
- Each program reasoning technique has its own pros and cons

<table>
<thead>
<tr>
<th>Method</th>
<th>Automatic</th>
<th>Sound</th>
<th>Complete</th>
<th>Object</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Program</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Assisted Proving</td>
<td>No</td>
<td>Yes</td>
<td>Yes/No</td>
<td>Model</td>
<td>Static</td>
</tr>
<tr>
<td>Model Checking of finite-state model</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Finite Model</td>
<td>Static</td>
</tr>
<tr>
<td>Model checking at program level</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Program</td>
<td>Static</td>
</tr>
<tr>
<td>Conservative Static Analysis</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Program</td>
<td>Static</td>
</tr>
<tr>
<td>Bug Finding</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Program</td>
<td>Static</td>
</tr>
</tbody>
</table>
OPLSS: Static Analysis

(1) Concepts in Static Analysis
(2) Operational / Denotational Semantics
(3) Abstract Interpretation
(4) Automatic Derivation of Static Analysis
Static Analysis for Android Multilingual Applications

• “Bittersweet ADB: Attacks and Defenses”
  ACM Symposium on Information, Computer and Communications Security, 2015 with Sungjae Hwang, Sungho Lee, and Yongdae Kim

• “All about Activity Injection: Threats, Semantics, and Detection”
  IEEE/ACM International Conference on Automated Software Engineering, 2017 with Sungho Lee and Sungjae Hwang

• “HybriDroid: Static Analysis Framework for Android Hybrid Applications”
  IEEE/ACM International Conference on Automated Software Engineering, 2016 with Sungho Lee and Julian Dolby

• “Towards Understanding and Reasoning about Android Interoperations”
  ACM/IEEE International Conference on Software Engineering, 2019 with Sora Bae and Sungho Lee

• “Adlib: Analyzer for Mobile Ad Platform Libraries”
  ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 with Sungho Lee
HybriDroid: Android Hybrid Apps

Android Java

Inter-language Communication

JavaScript
HybriDroid: Inter-language Communication

Android Java

JavaScript

Java Bridge

Web Browser

JavaScript Bridge
HybriDroid: Inter-language Communication

```
Class JSApp {
    @JavascriptInterface
    public int alert(String m) {
        ...
    }
    ...
}
...
addJavascriptInterface(
    new JSApp(), "app");
```

```
app.alert("hello hybrid");
```

Java Bridge

JavaScript Bridge
HybriDroid: Inter-language Communication

Android Java

```java
Class JSApp {
    @JavascriptInterface
    public int alert(String m) {
        ...
    }
}
...

addJavascriptInterface(
    new JSApp(), "app");
```

JavaScript

```javascript
app.alert("hello hybrid", 3);
```

MethodNotFound exception

Java Bridge

JavaScript Bridge
HybriDroid: Bug Detection

We found that hybrid app developers use bridge communication carefully without manipulating bridge objects and alarms. Surprisingly, all 24 true alarms are contained 24 true alarms and the other 5 apps contain 7 false positives, the causes of the bugs, and the time in seconds.

To evaluate the quality of Android hybrid apps in terms of number of unique bugs, the numbers of false positives and true positives, we collected all 48 hybrid apps using PlayDrone, a Google Play Store crawler [50].

In this section, we show the usefulness of HybriDroid and manually verified the reported bugs as summarized in Table 3. The first column presents the ranking groups, the numbers of unique bugs, the numbers of false positives and true positives.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Hybrid App</th>
<th>Bug Type (#)</th>
<th>#FP</th>
<th>#TP</th>
<th>Bug Cause (#)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – 100</td>
<td>com.gameloft.android.ANMP.GloftDMHM</td>
<td>MethodNotFound</td>
<td>0</td>
<td>1</td>
<td>Obfuscation (1)</td>
<td>2404 sec</td>
</tr>
<tr>
<td></td>
<td>com.creativemobile.DragRacing</td>
<td>MethodNotFound</td>
<td>1</td>
<td>0</td>
<td></td>
<td>3192 sec</td>
</tr>
<tr>
<td></td>
<td>com.gau.go.launcherex</td>
<td>MethodNotFound</td>
<td>2</td>
<td>0</td>
<td></td>
<td>5432 sec</td>
</tr>
<tr>
<td></td>
<td>com.tripadvisor.tripadvisor</td>
<td>MethodNotFound</td>
<td>0</td>
<td>1</td>
<td>Obfuscation (1)</td>
<td>4028 sec</td>
</tr>
<tr>
<td></td>
<td>com.dianxinos.dbxs</td>
<td>MethodNotFound</td>
<td>0</td>
<td>1</td>
<td>Obfuscation (1)</td>
<td>1924 sec</td>
</tr>
<tr>
<td>10,000 – 10,100</td>
<td>com.magemobile.game.LostWords</td>
<td>MethodNotFound</td>
<td>1</td>
<td>0</td>
<td></td>
<td>475 sec</td>
</tr>
<tr>
<td>20,000 – 20,100</td>
<td>com.daishin</td>
<td>MethodNotFound</td>
<td>0</td>
<td>1</td>
<td>Undeclared Method (1)</td>
<td>6572 sec</td>
</tr>
<tr>
<td>100,000 – 100,100</td>
<td>com.carezone.caredroid.careapp</td>
<td>MethodNotFound</td>
<td>0</td>
<td>5</td>
<td>Missing Annotation (5)</td>
<td>2357 sec</td>
</tr>
<tr>
<td></td>
<td>com.pateam.kanomthai</td>
<td>MethodNotFound</td>
<td>0</td>
<td>2</td>
<td>Missing Annotation (2)</td>
<td>4209 sec</td>
</tr>
<tr>
<td></td>
<td>com.acc5.16</td>
<td>MethodNotFound</td>
<td>0</td>
<td>6</td>
<td>Missing Annotation (6)</td>
<td>367 sec</td>
</tr>
<tr>
<td></td>
<td>jp.cleanup.android</td>
<td>MethodNotFound</td>
<td>1</td>
<td>0</td>
<td></td>
<td>253 sec</td>
</tr>
<tr>
<td></td>
<td>ligamexicana.futbol</td>
<td>MethodNotFound</td>
<td>2</td>
<td>0</td>
<td></td>
<td>253 sec</td>
</tr>
<tr>
<td>200,000 – 200,100</td>
<td>com.sysapk.weighter</td>
<td>MethodNotFound</td>
<td>0</td>
<td>1</td>
<td>Missing Annotation (1)</td>
<td>106 sec</td>
</tr>
<tr>
<td></td>
<td>com.youmustescape3guide.free</td>
<td>MethodNotFound</td>
<td>0</td>
<td>6</td>
<td>Missing Annotation (6)</td>
<td>445 sec</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>MethodNotFound</td>
<td>31</td>
<td>24</td>
<td></td>
<td>2287 sec</td>
</tr>
</tbody>
</table>

Even though bridge.receive();

```
@JavascriptInterface
String receive()
{
    ...
}
```

![Obfuscate] bridge.receive();

```
class JSApp{
    @JavascriptInterface
    String abc()
    {
        ...
    }

    bridge.receive();
```
Static Analysis for Android Multilingual Applications

• “Bittersweet ADB: Attacks and Defenses”
  ACM Symposium on Information, Computer and Communications Security, 2015 with Sungjae Hwang, Sungho Lee, and Yongdae Kim

• “All about Activity Injection: Threats, Semantics, and Detection”
  IEEE/ACM International Conference on Automated Software Engineering, 2017 with Sungho Lee and Sungjae Hwang

• “HybriDroid: Static Analysis Framework for Android Hybrid Applications”
  IEEE/ACM International Conference on Automated Software Engineering, 2016 with Sungho Lee and Julian Dolby

• “Towards Understanding and Reasoning about Android Interoperations”
  ACM/IEEE International Conference on Software Engineering, 2019 with Sora Bae and Sungho Lee

• “Adlib: Analyzer for Mobile Ad Platform Libraries”
  ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 with Sungho Lee
Inter-language Operation: Types and Values

Android Java

Class JSApp {
    @JavascriptInterface
    public int alert(String m) {
        ...
    }
    ...
}
...
addJavascriptInterface(new JSApp(), "app");

JavaScript

app.alert("hello hybrid");

Java Bridge

JavaScript Bridge
Inter-language Operation: Types and Values

Chapter 4. Types, Values, and Variables

The Java programming language is a *statically typed* language, which means that every variable and every expression has a type that is known at compile time.

The Java programming language is also a *strongly typed* language, because types limit the values that a variable ($4.12$) can hold or that an expression can produce.

The types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types ($4.2$) are the boolean type and special null type. An object ($4.3.1$) is a dynamically created instance of a class type or a dynamically created array. The values of a reference type are references to objects.

6 ECMAScript Data Types and Values

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further subclassified into ECMAScript types within this specification, the notation “Type(x)” is used as shorthand for “the type of x” where “type” refers to the ECMAScript language and specification types defined in this clause. When the term equivalent to saying “no value of any type”.

Android Java

JavaScript
Inter-language Operation: Overloading

8.4.9. Overloading
If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name.

This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the

When a method is invoked ($15.12$), the number of actual arguments (and any explicit type arguments) and the compile-time types of the arguments are
Formalization of Android Interoperation

- Identify the under-documented Android interoperation behaviors
  - Discovered previously-unknown, unintuitive, and surprising behaviors
- Present the first formal semantics of Android interoperation
- Develop a light-weight type system detecting interoperation bugs
  - More true bugs more efficiently than HybiDroid
OPLSS: Static Analysis

(1) Concepts in Static Analysis
(2) Operational / Denotational Semantics
(3) Abstract Interpretation
(4) Automatic Derivation of Static Analysis