Software Security VIII: Program Verification

Chengyu Song

Slides modified from David Wagner and Dawn Song
Administrivia

- Homework1
  - Due: today 11:59pm
  - Questions?
Finding vulnerabilities

- Dynamic analysis
  - Fuzzing
  - Symbolic execution
- Static analysis
Lattices in static analysis

A lattice is a set with
- a *partial order* for comparing elements
- a least upper bound called *join*
- a greatest lower bound called *meet*

In static analysis
- lattice elements abstract states
- order is used to check if results change
- meet and join are used at branch and join points
Most analyses use only meet or only join
Soundness and completeness

• Soundness: if the program contains an error, the analysis will report an error.
  • "Sound for reporting correctness"

• Completeness: if the analysis reports an error, the program will contain an error.
  • "Complete for reporting correctness"

Note: these terms have different meaning in other contexts
## Soundness and completeness (cont.)

<table>
<thead>
<tr>
<th></th>
<th>Complete</th>
<th>Incomplete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound</strong></td>
<td>Report all errors</td>
<td>Report all errors</td>
</tr>
<tr>
<td></td>
<td>Report no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td><strong>UNDECIDABLE</strong></td>
<td>(Ex: Abstract interpretation)</td>
</tr>
<tr>
<td></td>
<td>(Ex: manual verification)</td>
<td></td>
</tr>
<tr>
<td><strong>Unsound</strong></td>
<td>May not report all errors</td>
<td>May not report all errors</td>
</tr>
<tr>
<td></td>
<td>Report no false alarms</td>
<td>May report false alarms</td>
</tr>
<tr>
<td></td>
<td>(Ex: symbolic execution)</td>
<td>(Ex: Syntactic analysis)</td>
</tr>
</tbody>
</table>
Program verification

• Properties: true for every possible execution
  • Safety: nothing bad happens (e.g., buffer overflow)
  • Liveness: something good eventually happens

• Program verification in security
  • How to prove safety properties
How to reason about safety

• Approach: build up confidence on a function-by-function/module-by-module basis

• Modularity provides boundaries for our reasoning
  • Preconditions: what must hold for function to operate correctly
  • Postconditions: what holds after function completes

• These basically describe a contract for using the module
  • Most basic contract? Argument number and types
Functions in verification

- **Mathematical function**: \( f(x) \rightarrow y \)

- Individual statement can be considered as a function
  - Preconditions: what must hold for correctness of the statement
  - Postcondition: what holds after execution of the statement
  - Stmt #1’s postcondition should logically imply Stmt #2’s precondition

- **Invariants**: conditions that always hold at a given point in a function
Memory safety

- Memory access/dereference as a function

```c
byte deref(byte *p) {
    return *p;
}
```

- What is the precondition for the correctness of this function?
Memory safety (cont.)

• What is the precondition for the correctness of this function?

```c
/* p != NULL &&
   p does not point to freed object &&
   p does not point to uninitialized memory &&
   p is with the upper and lower bounds */
byte deref(byte *p) {
    return *p;
}
```
Verification (1)

- Proving precondition -> postcondition
- Given preconditions and postconditions
  - Specifying what obligations caller has (precondition) and what callers are entitled to rely upon (postcondition)
- Verify: no matter how function is called
  - If precondition is met at function's entrance
  - then postcondition is guaranteed to hold upon function's return
Verification (2)

Basic idea:

- Write down a precondition and postcondition for every line of code
- Use logical reasoning
Verification (3)

- Requirement
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between statements, write down invariants that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one
Example

• How to prove the following function won't have buffer overflow?

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access?
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* ?? */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires?
(3) Propagate requirement up to beginning of function
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++) {
        /* requires: a != NULL &&
           0 <= i && i < size(a) */
        total += a[i];
    }
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
           0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

```c
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: a != NULL &&
           0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

Let's simplify, given that `a` never changes.
Example

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
Example

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: 0 <= i && i < size(a) */
        total += a[i];
    return total;
}
```

Let's simplify given that the $0 \leq i$ part is clear.
Example

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++) {
        /* requires: i < size(a) */
        total += a[i];
    }
    return total;
}
```
Example

/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?
Example

```c
/* requires: a != NULL */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

How to prove our candidate invariant?

\( n \leq \text{size}(a) \) is straightforward because \( n \) never changes.
Example

```c
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```
Example

```c
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}

What about i < n ?
```
Example

```c
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

What about \( i < n \) ? That follows from the loop condition.
Example

/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}

At this point we know the proposed invariant will always hold...
Example

/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL &&
           0 <= i && i < n && n <= size(a) */
        total += a[i];
    return total;
}

... and we’re done!
Example

```c
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL &&
            0 <= i && i < n && n <= size(a) */
        total += a[i];
    return total;
}
```

A more complicated loop might need us to use induction:

**Base case**: first entrance into loop.

**Induction**: show that postcondition of last statement of loop plus loop test condition implies invariant.
Summary

• Software security: vulnerabilities
  • Exploits: the most popular way of getting attacked, including malware
  • Memory vulnerabilities: root causes, how to exploit, defense mechanisms
  • How to find vulnerabilities: fuzzing, symbolic execution, static analysis, verification
  • Other vulnerabilities?
    • In future sessions
For next class ...

- Midterm review