Dataflow Analysis: Class Analysis (conclusion)
Binary Analysis: PCode, SSA
Announcements

- HW3 is out
  - Standard homework: XTA
  - Alternative homework: Points-to over PCode
  - Please choose one and let me know by the end of week

- HW2 questions?
Announcements

- Office hours change
  - Mondays 4-5pm on Webex
  - Fridays 4-5pm on Webex
  - Mondays and Thursdays 2pm - in SAGE 3713
Outline of Today’s Class

- Class analysis, conclusion
  - RTA (last week)
  - XTA (last week)
  - 0-CFA
  - Points-to analysis (PTA)

- Introduction to Ghidra and PCode
- SSA
**XTA**

- **R** is the set of **reachable methods**
- **$S_m$** is the set of **types** that flow to method **m**
- **$S_f$** is the set of **types** that flow to field **f**

1. \{ main \} \subseteq R

2. for each method $m \in R$ and each **new site** new $C$ in $m$

   \{ C \} \subseteq S_m

   $\exists C \subseteq I$
3. for each method $m \in R$, each virtual call $y.n(z)$ in $m$, each class $C$ in $\text{SubTypes(StaticType}(y)) \cap S_m$ and $n'$, where $n' = \text{resolve}(C,n)$

(3.1) $\{n'\} \subseteq R$ // add $n'$ to $R$ if not already there
(3.2) $\{C\} \subseteq S_{n'}$ // add $C$ to $S_{n'}$ if not already there
(3.3) $S_m \cap \text{SubTypes(StaticType}(p)) \subseteq S_{n'}$, $S_w \subseteq S_{u'}$
(3.4) $S_{n'} \cap \text{SubTypes(StaticType}(\text{ret})) \subseteq S_m$, $S_{u'} \subseteq S_m$

($p$ denotes the parameter of $n'$, and $\text{ret}$ denotes the return of $n'$)
4. for each method \( m \in R \), each field read \( x = y.f \) in \( m \)

\[ S_f \subseteq S_m \]

5. for each method \( m \in R \), each field write \( x.f = y \) in \( m \)

\[ S_m \cap \text{SubTypes(StaticType}(f)) \subseteq S_f \]
Practical Concerns

- Multiple parameters
- Direct calls
  - either static invoke calls or special invoke calls
- Array reads and writes!
- Static fields
- See Tip and Palsberg for more
public class A {
    public static void main() {
        n1();
        n2();
    }
    static void n1() {
        A a1 = new B();
        a1.m();
    }
    static void n2() {
        A a2 = new C();
        a2.m();
    }
}

RTA:
\[ T = \{ B, C \} \]
\[ a1 : \{ B, C \} \]
\[ a1.m() : B.m(), C.m() \]

XTA:
\[ S_{m_1} = \{ B \} \]
\[ S_{m_2} = \{ C \} \]
\[ a1 : \{ B \} \]
\[ a1.m() : \exists B.m() \]
public class OrExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;

    public OrExp(BoolExp left, BoolExp right) {
        this.left = left;
        this.right = right;
    }

    public boolean evaluate(Context c) {
        private BoolExp l = this.left;
        private BoolExp r = this.right;
        return l.evaluate(c) || r.evaluate(c);
    }
}

Boolean Expression Hierarchy: RTA vs. XTA vs. “Ground Truth”
main() {
    Context theContext = new Context();
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp exp = new AndExp(new Constant(true), new OrExp(x, y));
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}
Outline of Today’s Class

- Class analysis, conclusion
  - RTA (last week)
  - XTA (last week)
  - 0-CFA
  - Points-to analysis (PTA)
- Introduction to Ghidra and PCode
- SSA
0-CFA

- Described in Tip and Palsbserg’s paper

- 0-CFA stands for 0-level Control Flow Analysis, where “0-level” stands for context-insensitive analysis
  - Will see 1-CFA, 2-CFA, … k-CFA later

- Improves on XTA by storing even more information about flow of class types
0-CFA

\( R \) is the set of \textit{reachable methods}

\( S_v \) is the set of \textit{types} that flow to \textit{variable} \( v \)

\( S_f \) is the set of \textit{types} that flow to field \( f \)

1. \{ \text{main} \} \subseteq R

2. for each method \( m \in R \) and each \textit{new site} \( x = \text{new} \ C \) in \( m \)
   \{ C \} \subseteq S_x
3. for each method $m \in R$, each virtual call $x = y.n(z)$ in $m$, each class $C$ in $S_y$ and $n'$, where $n' = \text{resolve}(C,n)$

\begin{align*}
(3.1) \{ n' \} & \subseteq R \\
(3.2) \{ C \} & \subseteq S_{\text{this}} \\
(3.3) S_z \cap \text{SubTypes}(\text{StaticType}(p)) & \subseteq S_p \\
(3.4) S_{\text{ret}} \cap \text{SubTypes}(\text{StaticType}(x)) & \subseteq S_x
\end{align*}

(this is the implicit parameter of $n'$, $p$ is the parameter of $n'$, and $\text{ret}$ is the return of $n'$)
0-CFA

4. for each method $m \in R$, each field read $x = y.f$ in $m$

$$S_x \cap \text{SubTypes}(\text{StaticType}(x)) \subseteq S_x$$

5. for each method $m \in R$, each field write $x.f = y$ in $m$

$$S_y \cap \text{SubTypes}(\text{StaticType}(f)) \subseteq S_f$$
6. for each method $m \in R$, each assignment $x = y$ in $m$,

$$S_y \cap \text{SubTypes}(\text{StaticType}(x)) \subseteq S_x$$
Example: XTA vs. 0-CFA

```java
public class A {
    public static void main() {
        A a1 = new B();
        a1.m();
        A a2 = new C();
        a2.m();
    }
}
```

**XTA:**
- `a1.m(): B.m(), C.m()`
- `a2.m(): B.m(), C.m()`

**SA1:**
- `a1.m() ∈ {B, C}`

**0-CFA:**
- `a1.m() ∈ {B}`
- `a2.m(): B.m()`

Diagram:
```
A -> B -> m() -> C -> m()
```

```
XTA: a1 = B, C; SA1: a1 = B; 0-CFA: a1 = B
```
public class OrExp extends BoolExp {
    private BoolExp left;
    private BoolExp right;

    public OrExp(BoolExp left, BoolExp right) {
        this.left = left;
        this.right = right;
    }

    public boolean evaluate(Context c) {
        private BoolExp l = this.left;
        private BoolExp r = this.right;
        return l.evaluate(c) || r.evaluate(c);
    }
}
main() {
    Context theContext = new Context();
    BoolExp x = new VarExp("X");
    BoolExp y = new VarExp("Y");
    BoolExp exp = new AndExp(
        new Constant(true), new OrExp(x, y)
    );
    theContext.assign(x, true);
    theContext.assign(y, false);
    boolean result = exp.evaluate(theContext);
}

Boolean Expression Hierarchy:
XTA vs. 0-CFA
Outline of Today’s Class

- Class analysis, conclusion
  - RTA (last week)
  - XTA (last week)
  - 0-CFA
  - Points-to analysis (PTA)

- Introduction to Ghidra and PCode
- SSA
PTA

- Widely referred to as Andersen’s points-to-analysis for Java

- Improves on 0-CFA by storing information about **objects**, not classes

```java
A a1 = new A(); // o₁
A a2 = new A(); // o₂

a1.f = new B();
a2.f = new C();
x = a1.f
```

```
PTA

0-CFA
A.f ∈ B, C₂
x ∈ B, C₂

0₁.f ∈ B
0₂.f ∈ C
```

CSCI 4450/6450, A Milanova
PTA

\( \mathcal{R} \) is the set of reachable methods

\( \mathcal{P}_t(v) \) is the set of objects that \( v \) may point to

\( \mathcal{P}_t(x) \) is the set of objects that field \( f \) of object \( o \) may point to

1. \( \{ \text{main} \} \subseteq \mathcal{R} \)

2. for each method \( m \in \mathcal{R} \) and each new site \( i \):
   \[ x = \text{new } C \text{ in } m \]
   \( \{ o_i \} \subseteq \mathcal{P}_t(x) \)

   // instead of \( C \), we have \( o_i \)
3. for each method $m \in R$, each virtual call $x = y.n(z)$ in $m$, each class $o_i$ in $\text{Pt}(y)$ and $n'$, where $n' = \text{resolve}(\text{class}_\text{of}(o_i), n)$

\[(3.4) \{ n' \} \subseteq R\]
\[(3.2) \{ o_i \} \subseteq \text{Pt}(\text{this})\]
\[(3.3) \text{Pt}(z) \cap \text{SubTypes}(\text{StaticType}(p)) \subseteq \text{Pt}(p)\]
\[(3.4) \text{Pt}(\text{ret}) \cap \text{SubTypes}(\text{StaticType}(x)) \subseteq \text{Pt}(x)\]

(this is the implicit parameter of $n'$, $p$ is the parameter of $n'$, and $\text{ret}$ is the return of $n'$)
4. for each method \( m \in R \), each field read \( x = y.f \) in \( m \)

   for each object \( o \in \text{Pt}(y) \)

\[
\text{Pt}(o.f) \cap \text{SubTypes}(\text{StaticType}(x)) \subseteq \text{Pt}(x)
\]

5. for each method \( m \in R \), each field write \( x.f = y \) in \( m \)

   for each object \( o \in \text{Pt}(x) \)

\[
\text{Pt}(y) \cap \text{SubTypes}(\text{StaticType}(f)) \subseteq \text{Pt}(o.f)
\]
6. for each method $m \in R$, each assignment stmt $x = y$ in $m$

$$\text{Pt}(y) \cap \text{SubTypes(StaticType}(x)) \subseteq \text{Pt}(x)$$
Example: 0-CFA vs. PTA

```java
public class A {
    public static void main() {
        X x1 = new X();    // o1
        A a1 = new B();   // o2
        x1.f = a1;  // o1.f points to o2
        A a2 = x1.f;  // a2 points to o2
        a2.m();

        X x2 = new X();    // o3
        A a3 = new C();   // o4
        x2.f = a3;  // o3.f points to o4
        A a4 = x2.f;  // a4 points to o4
        a4.m();
    }
}
```
The Big Picture

- All fit into our monotone dataflow framework!
- Flow-insensitive, context-insensitive
  - Compute single solution $S$
- Algorithms differ mainly in “size” of $S$
  - RTA: only 2 kinds of statements; Lattice?
  - XTA: expands to all statements; Lattice?
  - 0-CFA: all statements; Lattice?
  - PTA (Points-to analysis): all statements; Lattice elements are points-to graphs
The Big Picture

RTA:

Types: A B C D

XTA:

S_{m1} S_{m2} ... S_{mk} S_{f1} ... S_{fk}

0-CFA:

v_1, v_2, ... v_n

PTA:

v_1, v_2, ... v_n

A B C D ...

o_1:A o_2:A o_3:B o_4:B o_5:C o_6:D ...

CSCI 4450/6450, A Milanova
Outline of Today’s Class

- Class analysis, conclusion
  - RTA (last week)
  - XTA (last week)
  - 0-CFA
  - Points-to analysis (PTA)

- Introduction to Ghidra and PCode
- SSA
Ghidra: https://ghidra-sre.org/

- An open-source reverse engineering tool
- Functionality similar to Binary Ninja and IDA
Ghidra Basics

- **Ghidra Project** – a collection of binaries and libraries under analysis
- **Ghidra Program** – an individual binary under analysis
- **Ghidra Tool** – an analysis tool
  - **Code Browser**
- **Ghidra Script** – a static analysis on top of Ghidra API; this is what we’ll do
Ghidra Basics

- A reverse engineering/decompilation tool
  - Builds a high-level program representation from x86 (and other?) binaries

- A program analysis framework
  - Exposes API into program representation
    - Similarly to the way Soot exposes API over Jimple

- All “perks” of program analysis frameworks
  - Huge API, not well documented, buggy
Intermediate Representation (IR)

Compiler (roughly):

Source \(\rightarrow\) AST \(\rightarrow\) 3-address Code \(\rightarrow\) ASM \(\rightarrow\) Machine Code

Ghidra is a Decompiler:

Machine Code (x86) \(\rightarrow\) \(\ldots\) \(\rightarrow\) 3-address Code (PCode) \(\rightarrow\) \(\ldots\) \(\rightarrow\) Source???
```c
void fill_bowl(char* ingredients, char* bowl, int bowl_size)
{
    printf("How many ingredients?\n");

    int number;
    scanf("%u", &number);

    if (number > bowl_size)
        number = bowl_size - 1;

    // Copy at most bowl_size characters into the buffer
    for (int i=0; i <= number; i++)
    {
        bowl[i] = ingredients[i];
    }
}
```
void fill_bowl(char *ingredients, char *bowl, int bowl_size)
{
    int local_24, local_20, i, number;
    i = bowl_size;
    _printf("...");
    _scanf("%u", &local_20);
    if (i < local_20) {
        local_20 = i + -1;
    }
    local_24 = 0;
    while (local_24 <= local_20) {
        bowl[local_24] = ingredients[local_24];
        local_24 = local_24 + 1;
    }
    return;
}
fill_bowl IR and CFG

1. local_20 = USER_INPUT
2. i = bowl_size

3. if i < local_20
   T
   4. local_20 = i - 1
5. local_24 = 0

6. if local_24 <= local_20
   T
   7. t1 = ingredients[local_24]
   8. bowl[local_24] = t1
   9. local_24 = local_24 + 1

10. ...
**PCode**

- **Varnode** – a “variable”, i.e., a named location
  - A register, stack location, constant value (r-value), global variable, ...
  - E.g., \((\text{stack}, 0xfffffffffffffffe0, 4)\)
- **PCodeOp** is a 3-address code instruction
  - Reversed from x86, lower-level than Jimple
    - \((\text{unique}, 0x3100, 8) \quad \text{INT ADD} \quad \text{(ram, 0x100002078, 8)} , \quad \text{(const, 0x8, 8)}\)
    - \((\text{unique}, 0x3100, 8)\) – stack location (l-value), a pointer
    - \((\text{ram, 0x100002078, 8})\) – server global (r-value)
    - \((\text{const, 0x8, 8})\) – offset into server struct
  - Roughly: \(\text{tmp} = \text{server} + \text{field_offset}\)
PCode

- **PCodeOp instruction**
  - **Output**: as l-value
    
  - **Opcode**
    
  - **Inputs**: as r-values
    
- *(unique, 0x3100, 8) INT_ADD (ram, 0x100002078, 8), (const, 0x8, 8)*

- tmp = server+field_offset

- *(unique, 0xc080, 8) LOAD (const, 0x1b1, 4), (unique, 0x3100, 8)*

- tmp2 = *tmp
Static Single Assignment (SSA)

- SSA form transforms the 3-address code of a program so that each variable is defined exactly once (statically)
  - A sparse, efficient representation
  - Universally applied technique in compilers and static analysis
- Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”, TOPLAS 1991
Multiple definitions for a given use

- E.g., def-use chains for `local_24`: (5,6) and (9,6)
- Inconvenient, potentially expensive (quadratic)
Easy case

\[ x = \text{INPUT}; \]
\[ y = x + 10; \]
\[ x = x + y; \]

becomes

\[ x_1 = \text{INPUT}; \]
\[ y = x_1 + 10; \]
\[ x_2 = x_1 + y; \]

The 2 definitions of \( x \) become definitions of \( x_1 \) and \( x_2 \) respectively.
Phi Functions

If-then-else

\[
\begin{align*}
\text{if } (a>0) & \quad \text{if } (a>0) \\
x &= 5; & x_1 &= 5; \\
\text{else } & \quad \text{becomes else} \\
x &= 10; & x_2 &= 10; \\
& \quad x_3 = \phi(x_1, x_2)
\end{align*}
\]

Phi function introduces a (static) definition of \(x, x_3\). If control took True arm, then \(x_3\) is \(x_1\), otherwise it is \(x_2\).
Phi Functions

Loops

\[
x = 0;
\]

while (x <= n)

\[
x = x + 1;
\]

\[
x1 = 0;
\]

\[
x2 = \phi(x1, x3)
\]

while (x2 <= n)

\[
x3 = x2 + 1;
\]

If control took forward edge, \( x2 \) is \( x1 \); otherwise, i.e., control took back edge, it is \( x3 \).
Now there is a single def-use chain for \texttt{local\_24\_2}: (6,7)
PCode Uses SSA

- Essential classes and methods:
  - `PcodeOp`, `Varnode`, `PcodeBlockBasic`
  - `var.getDef()` retrieves the single definition of `var`

- Phi-nodes: `MULTIEQUAL`

- `(unique ... ) varnodes?
Outline of Today’s Class

- Class analysis, conclusion
  - RTA (last week)
  - XTA (last week)
  - 0-CFA
  - Points-to analysis (PTA)

- Introduction to Ghidra and PCode
- SSA