Dynamic Dataflow Analysis

CS/SWE 795, Fall 2017
Program Analysis for Software Testing

GEORGE MASON UNIVERSITY

Today

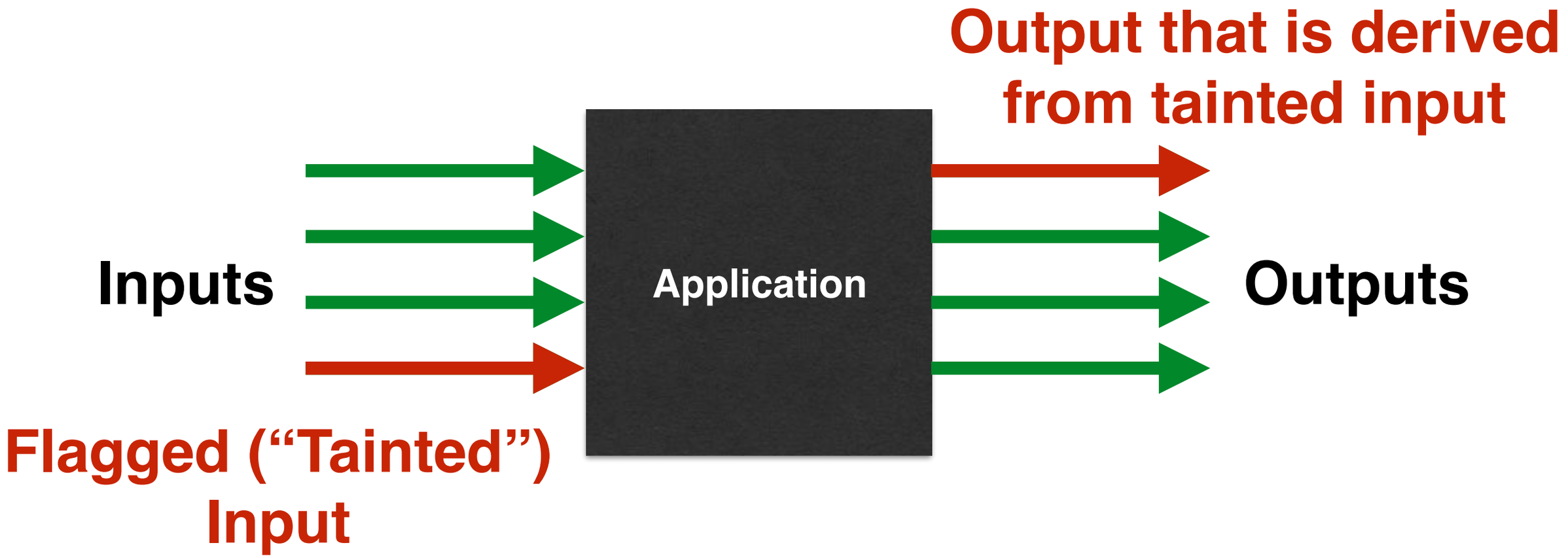
- HW 1 discussion
- DyTAN
- Phosphor
- Discussion of taint tracking and data flow analysis
- Phosphor Lab

HW 1 Discussion

Solution online at:
https://github.com/gmu-cs-795-f17/homework-1-jon-bell

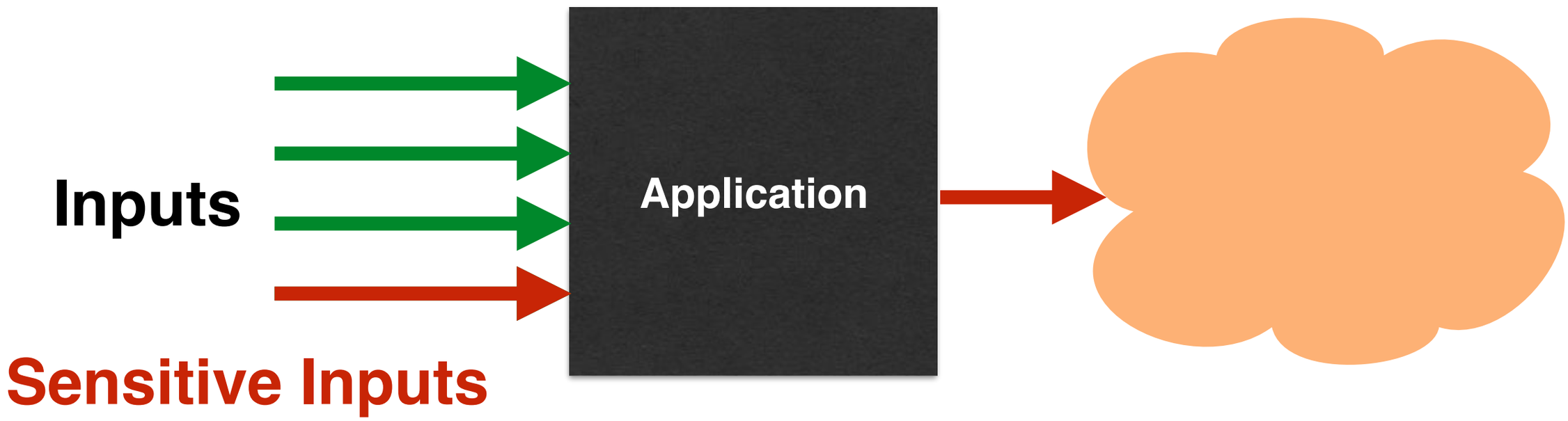
More info about frames:
https://stackoverflow.com/questions/20391272/understanding-how-to-use-visitframe?answertab=votes#tab-top

Dynamic Data Flow Analysis: Taint Tracking



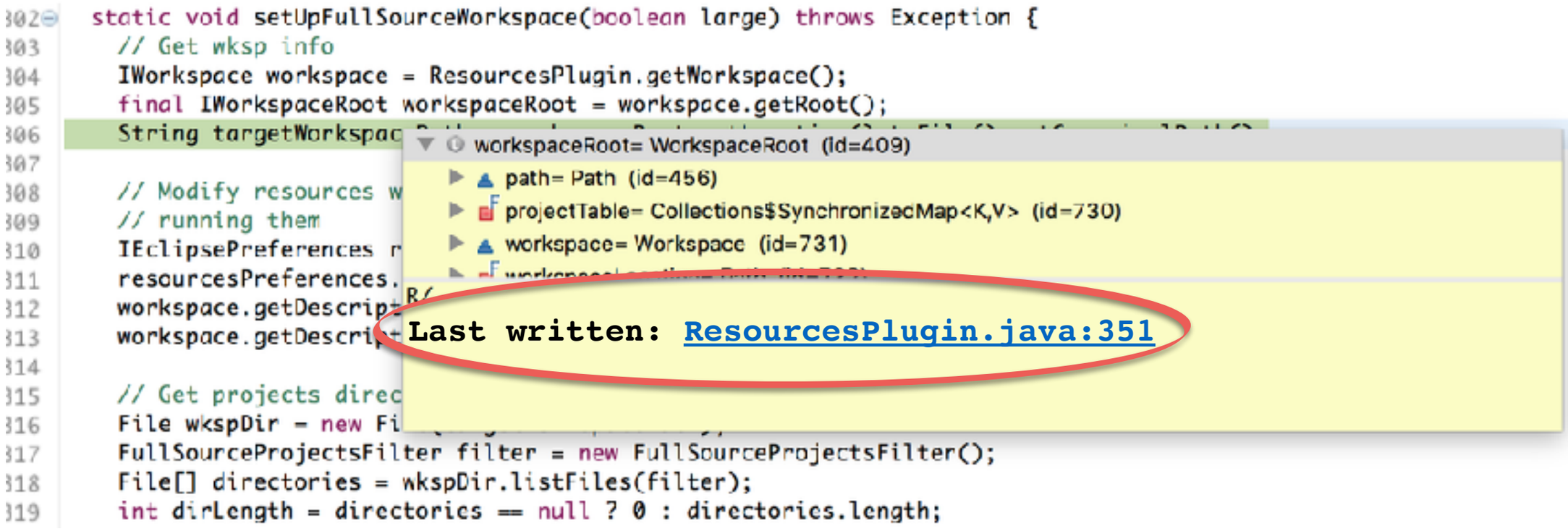
Taint Tracking: Applications

End-user privacy testing: Does this application send my personal data to remote servers?



Taint Tracking: Applications

Debugging: Which inputs are relevant to the current (crashed) application state? Where was a variable last written?



Taint Tracking: Applications

Testing: Are my test cases overly specified?

```
@Test
public void testEnrolled() throws Exception {
   Student s = new Student();
   s.id = 5;
   s.name = "Bob";
   s.setEnrolled();
   assertTrue(s.isEnrolled);
}
```

No assertion depends on these values

Taint Tracking Approaches

- Associate tags with data, then propagate the tags
- Approaches:
  - Operating System modifications [Vandebogart ’07], [Zeldovich ’06]
  - Language interpreter modifications [Chandra ’07], [Enck ’10], [Nair ’07], [Son ’13]
  - Source code modifications [Lam ‘06], [Xu ’06]
  - Binary instrumentation of applications [Clause ’07], [Cheng ’06], [Kemerlis ’12]

Not portable

Hard to be sound, precise, and performant

Dytan

(Discuss paper)

Phosphor

(Discuss paper, plus a few slides supporting technical bits)

Phosphor: Instrumentation Strategy

```
double pie = 3.14;
double more = 1;
double more_pie = pie + more;
int ret = callSomeMethod(pie);
```

```
double pie = 3.14;
int pie_tag = 0;
double more = 1;
int more_tag = 0;
double more_pie = pie + more;
int more_pie_tag = pie_tag | more_tag;
TaintedInt tmp = callSomeMethod(pie_tag, pie);
int ret = tmp.val;
int ret_tag = tmp.tag;
```

(Of course, we do this all at byte code, not source code)

Phosphor's Taint Tag Storage

	Local variable	Method argument	Return value	Operand stack	Field
Object	Stored as a field of the object				
Object array	Stored as a field of each object				
Primitive	Shadow variable	Shadow argument	"Boxed"	Below the value on stack	Shadow field
Primitive array	Shadow array variable	Shadow array argument	"Boxed"	Array below value on stack	Shadow array field

Taint Propagation

- Modify all byte code instructions to be taint-aware by adding extra instructions
- Examples:
  - Arithmetic -> combine tags of inputs
  - Load variable to stack -> Also load taint tag to stack
  - Modify method calls to pass taint tags

Complications

- Primitives on the stack
- Non-modifiable classes (e.g. Object, StackTraceElement, Byte, some others)
- Arrays of primitive values
  - And multi-dimensional arrays, and upcasting arrays
- Native methods

Phosphor Doubles Stack Size

Code Snippet

```
void foo(int i, int j)
{
 int k = i + j;
}
```

Bytecode

```
ILOAD 1
ILOAD 2
IADD
ISTORE 3
```

Code Snippet (instrumented)

```
void foo(int i_tag, int i,
   int j_tag, int j)
{
 int k = i + j;
 int k_tag = i_tag | j_tag;
}
```

Instrumented Bytecode

```
ILOAD 1
ILOAD 2
ILOAD 3
ILOAD 4
DUP2_X1
POP2
IADD
SWAP
IOR
ISTORE 6
ISTORE 5
```

Methods can get long, but avoids need for expensive shadow stack

Non-Modifiable Classes (and Arrays)

- Candidate approach:
  - Use a HashMap, with each untrackable object as key
  - Very, very slow (need to access a globally-locked HashMap for EVERY operation you do involving an object)
- Phosphor:
  - Special case everything
  - Primitive arrays: get their own shadow array, tracked through upcasting
  - Non-modifiable classes: HashMap

Challenge 1: Upcasting

```
byte[] array = new byte[5];
Object ret = array;
```

```
int[] array_tag = new int[5];
byte[] array = new byte[5];
Object ret = new TaintedByteArray(array,array_tag);
```

Solution 1: Box taint tag with array when upcasting

```
byte[] foo = (byte[]) ret;
```

```
int[] foo_tag = ((TaintedByteArray) ret).tag;
byte[] foo_tag = ((TaintedByteArray) ret).val;
```

Challenge 1: Upcasting

```
byte[] array = new byte[5];
byte[][] ret = new byte[]{array};
```

```
int[] array_tag = new int[5];
byte[] array = new byte[5];

TaintedByteArray[] ret = new TaintedByteArray[1];
ret[0] = new TaintedByteArray(array_tag,array);
```

```
byte[] foo = ret[0];
```

```
int[] foo_tag = ret[0].tag;
byte[] foo_tag =ret[0].val;
```

Challenge: Native Code

We can’t instrument everything!

Native Code

```
public int hashCode() {
  return super.hashCode() * field.hashCode();
}
```

```
public native int hashCode();
```

What caller expects → `public TaintedInt hashCode();`

Solution: Wrappers. Rename *every* method, and leave a wrapper behind. Always call wrapper version.

```
public TaintedInt hashCode$$wrapper() {
    return new TaintedInt(0, hashCode());
}
```

Native Code

Wrappers work both ways: native code can still call a method with the old signature

```
public int[] someMethod(byte in)
  {
    return someMethod$$wrapper(0, in).val;
  }


public TaintedIntArray someMethod$$wrapper(int in_tag, byte in)
{
  //The original method "someMethod", but with taint tracking
}
```

Native Code

- The main design limitation
- Return value’s tag becomes combination of all parameters (heuristic); not found to be a problem in our evaluation
- Note: reflection is NOT a limitation, is easiest of all challenges to work around (runtime wrappers)

Configuration Options

- Tag propagation modes:
  - Data flow
  - Control flow

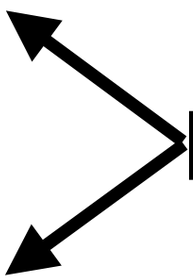
```
int c = a + b;
```

```
if (a == 0) c = 0;
```

- Tag format:
  - Integer (bit vectors)
  - Object (maintain relationships sets)
- Automatic Tagging and Checking

Implicit vs. Explicit Data Flow

```
String secretStr ="secret";
String leakedStr = "";
switch(secretStr.charAt(i))
{
  case 'a':
    leakedStr += 'a';
    break;
  case 'b':
    leakedStr += 'b';
    ...
}
```



No “explicit” data flow!

Phosphor: API

Getting and setting tags on objects

```
Interface TaintedWithObjTag.class
public Taint getPHOSPHOR_TAG();
public void setPHOSPHOR_TAG(Object o);
```

Getting and setting tags on primitives

```
MultiTainter.class
public static Taint getTaint(<Primitive Type> c);
public static float tainted<Primitive Type>(<Primitive Type> f, Object tag);
```

Getting relationships between tags

```
Class Taint.class
public LinkedList<Taint> getDependencies();
public Object getLabel();
```

Dynamic Dataflow Applications

(Discuss)

Lab: Phosphor