

# Modelgen:

Mining Explicit Information Flow

Specifications from Concrete Executions



**Lazaro Clapp, Saswat Anand, Alex Aiken**

Stanford University



Why mine specifications?

# Whole-program static analysis

**Application**



# Whole-program static analysis



**Malware?**

**Bugs?**

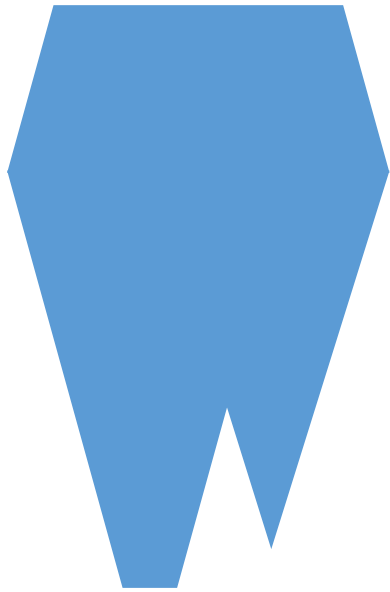
**Documentation**

# Whole-program static analysis?

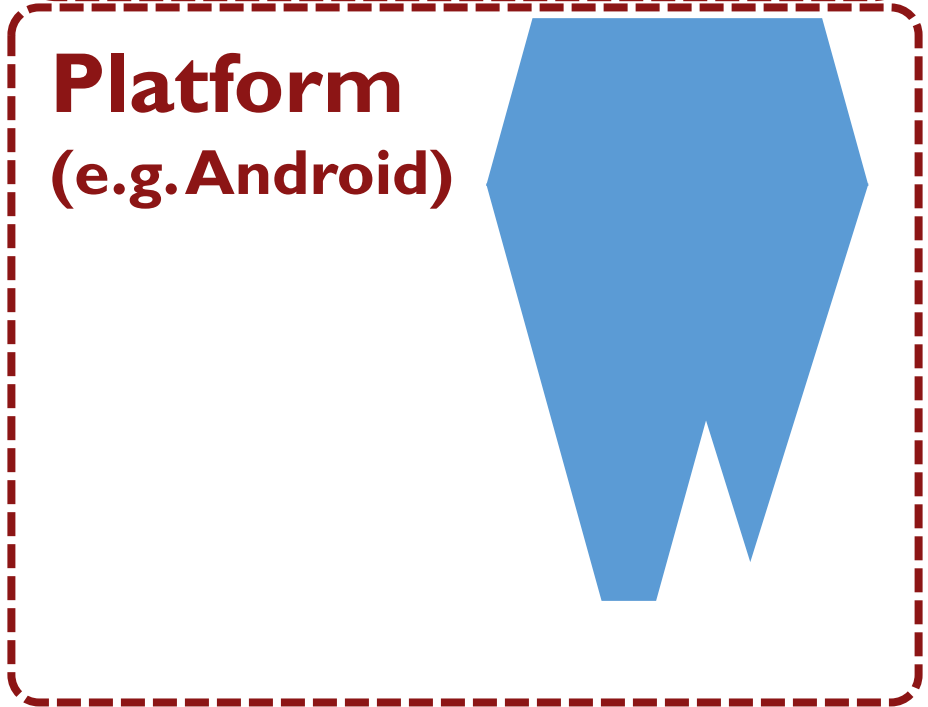
**Application**



**Platform**  
(e.g. Android)



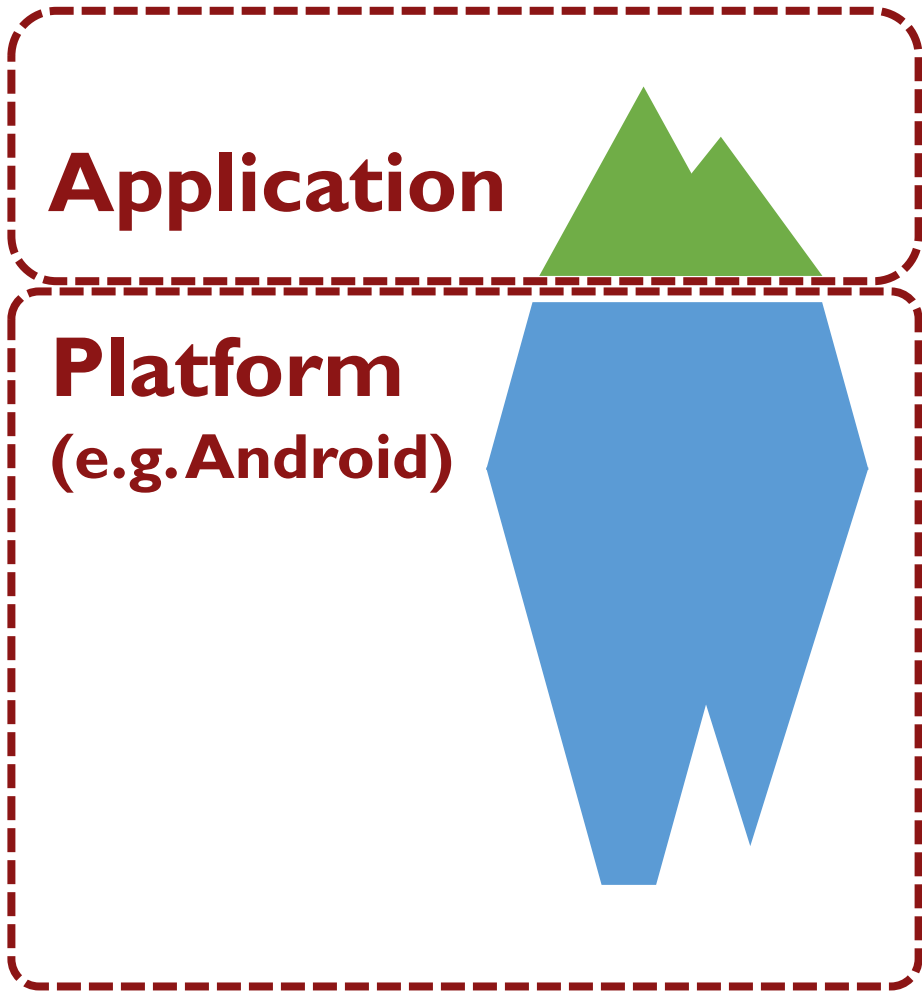
# Whole-program static analysis?



**???**

Three large, bold, dark red question marks.

# Whole-program static analysis?



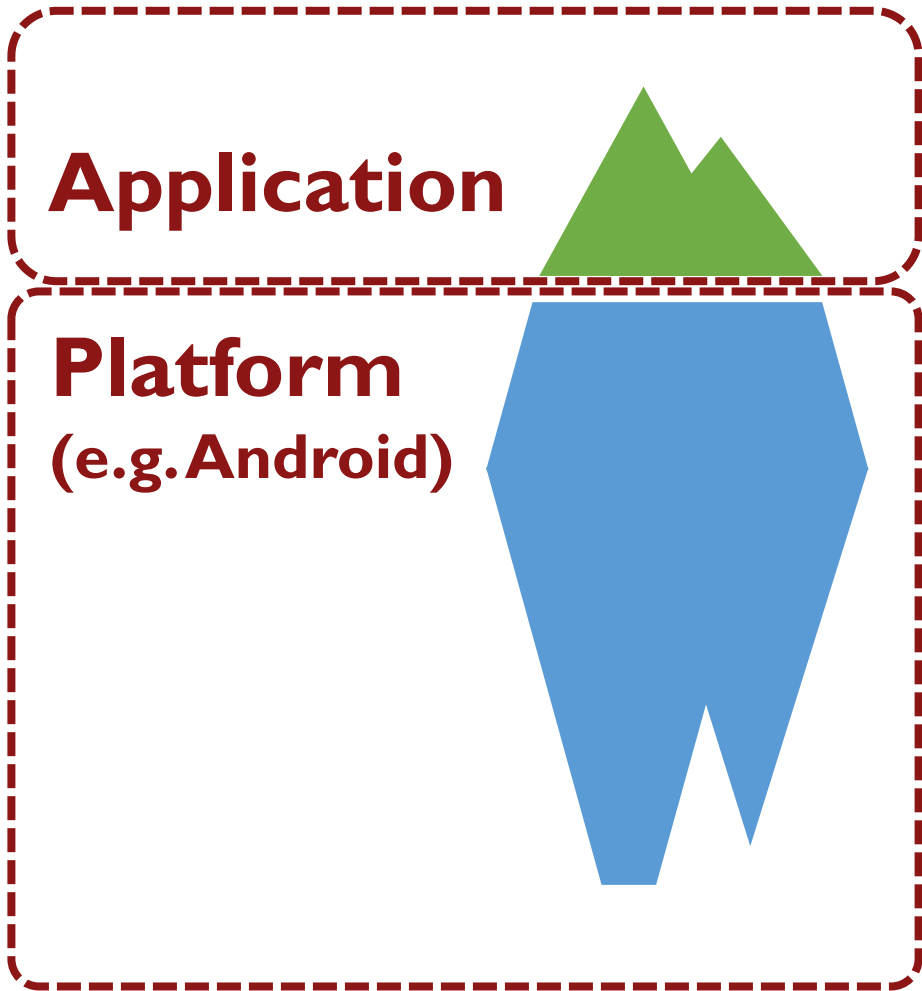
Static  
Analysis



???

- **Native code**

# Whole-program static analysis?



Static  
Analysis

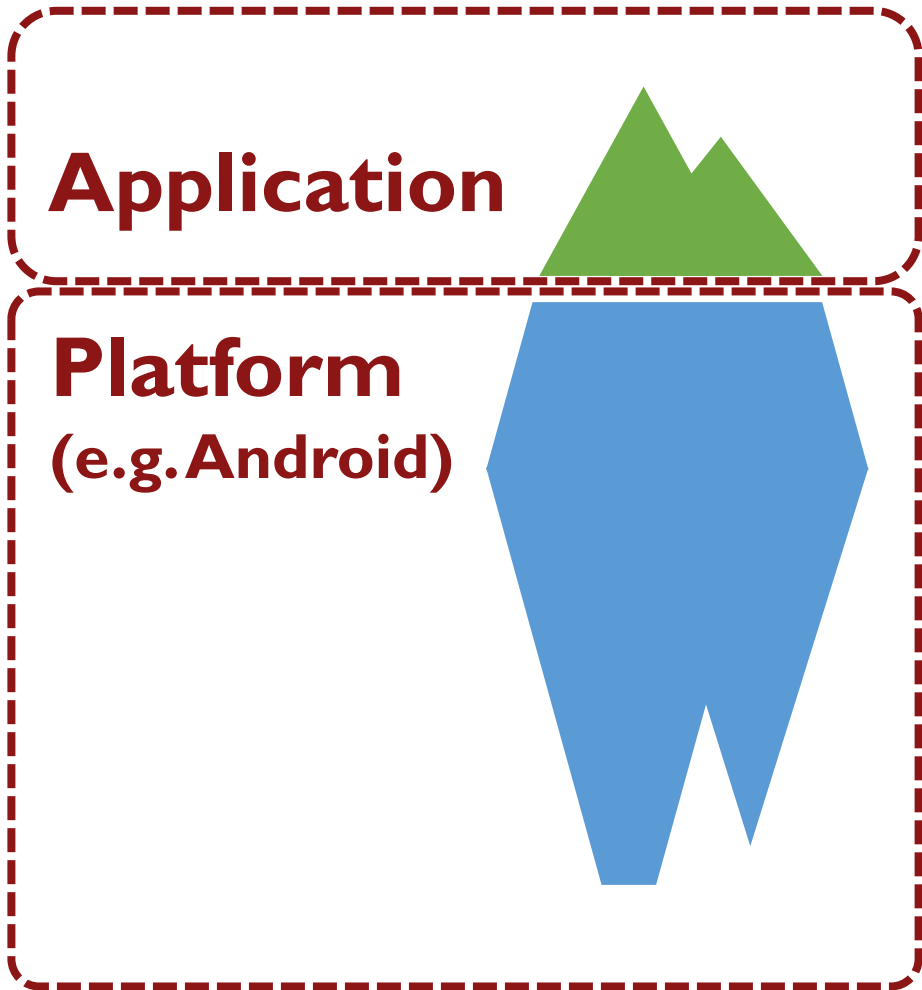


???

- **Native code**
- **Reflection**



# Whole-program static analysis?



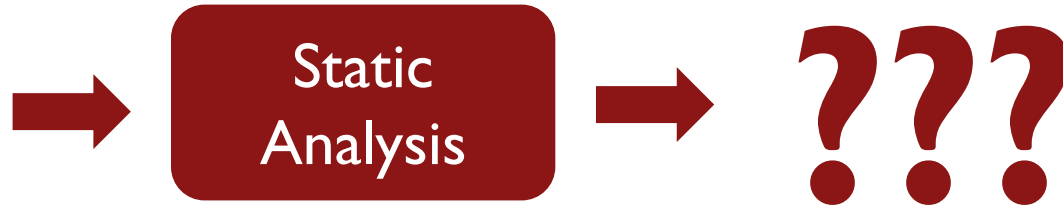
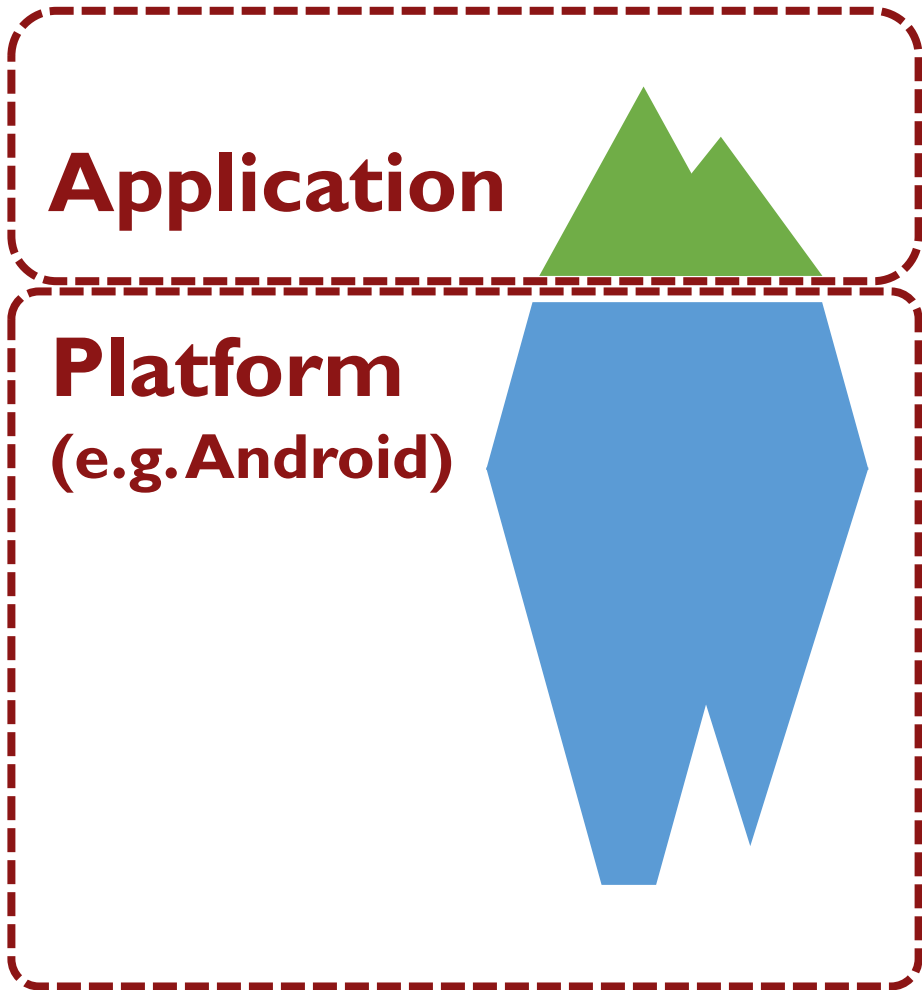
Static  
Analysis



???

- **Native code**
- **Reflection**
- **Complex OOP patterns / indirection**

# Whole-program static analysis?



- **Native code**
- **Reflection**
- **Complex OOP patterns / indirection**
- **Large (e.g. Android >2 MLOC, Java)**

# Whole-program static analysis?



Static Analysis

???

**Hard to Analyze Statically**

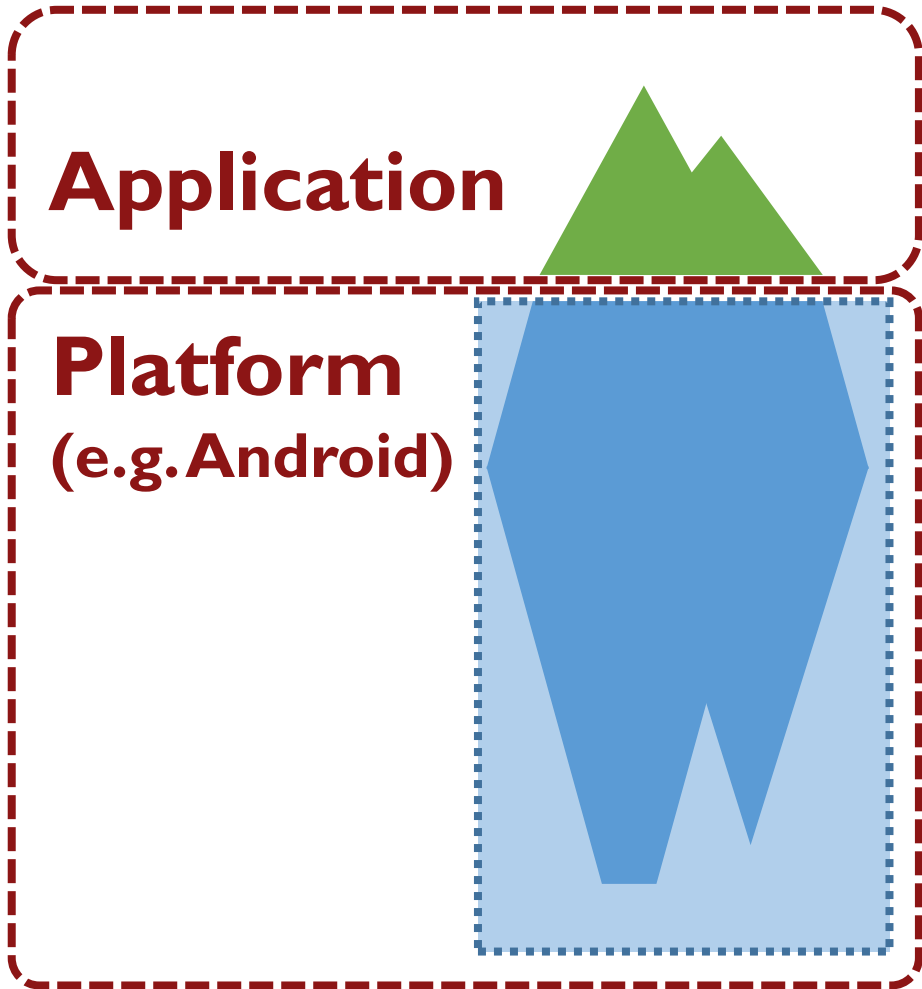
- Complex control flow direction
- Large (e.g. Android >2 MLOC, Java)

# Options: Best-case



**Under-approximation**  
**(Very) Unsound**  
**False negatives**

# Options: Worst-case

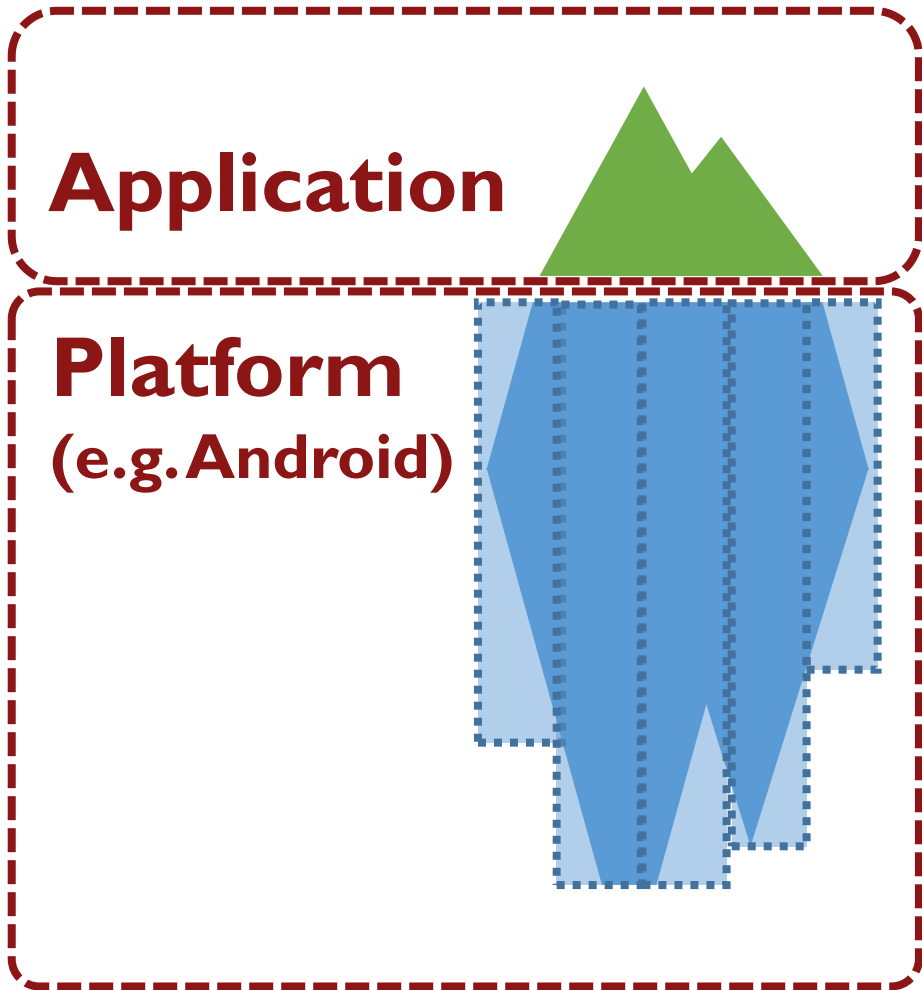


Static Analysis



**Over-approximation**  
**(Very) Imprecise**  
**False positives**

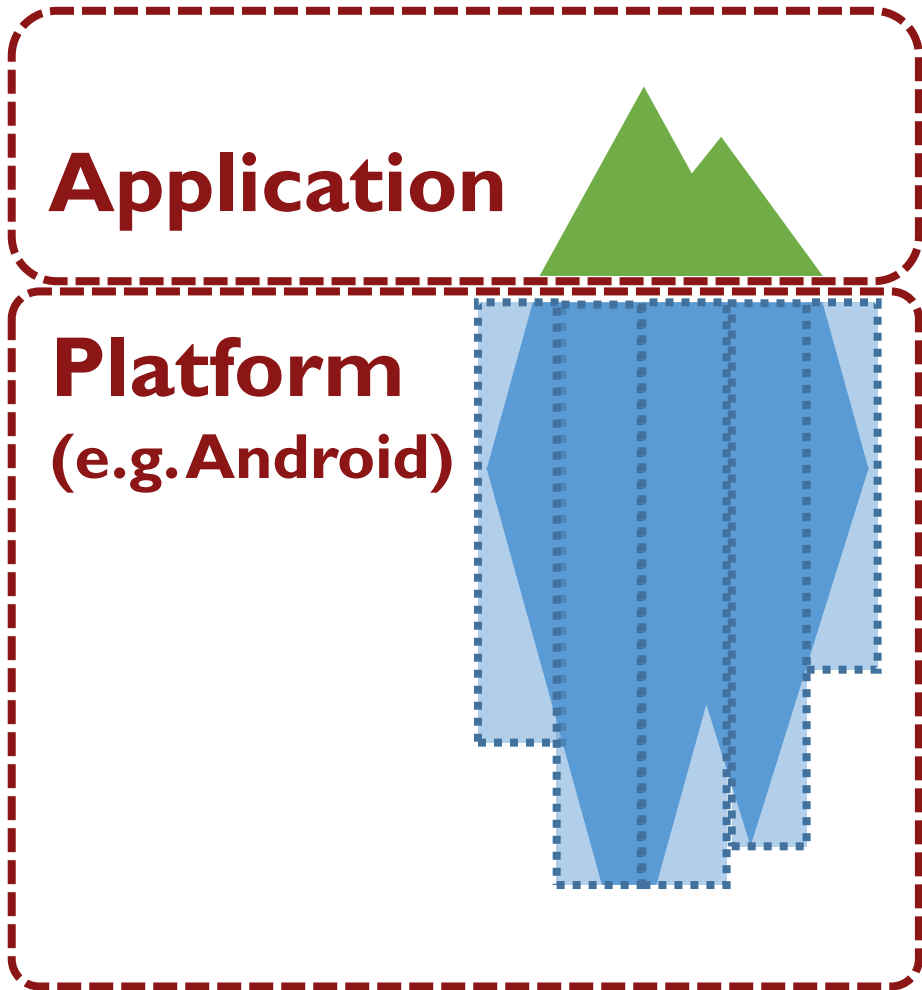
# Options: Specifications



- **Slight over-approximation**
- **Manually written**
- **Effort intensive\***

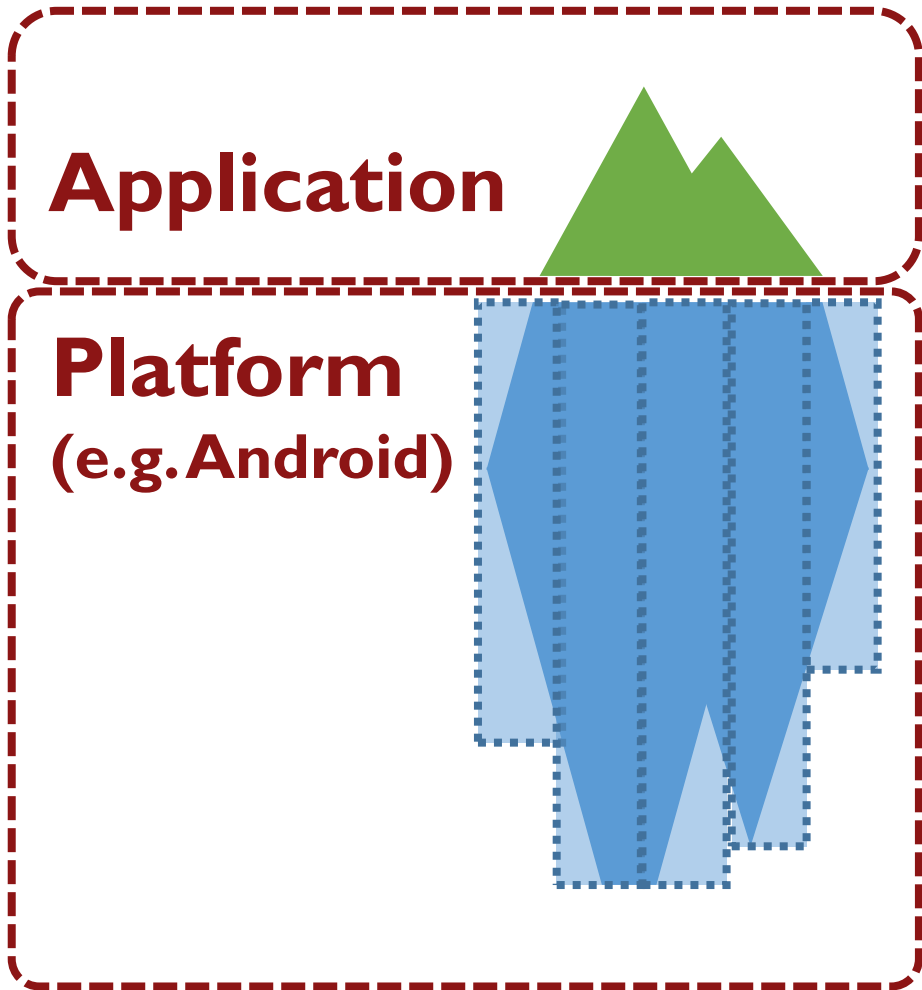
\* Our system (STAMP):  
Models for 1,116 methods, written over 2 years

# Mining Specifications



- **Slight over-approximation**
- ~~**Manually written**~~
- ~~**Effort intensive**~~

# Mining Specifications



- **Slight over-approximation**
- **Mined automatically using dynamic analysis**

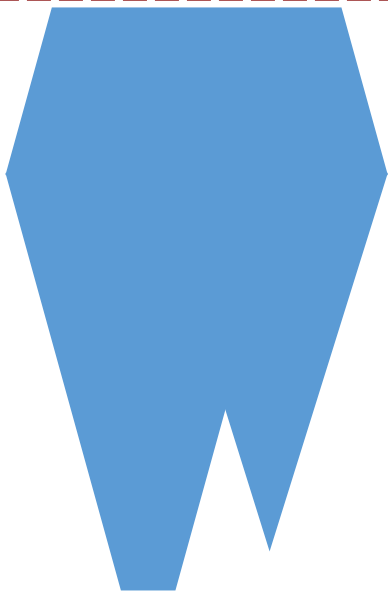


# Mining Specifications

**Application**



**Platform**  
(e.g. Android)

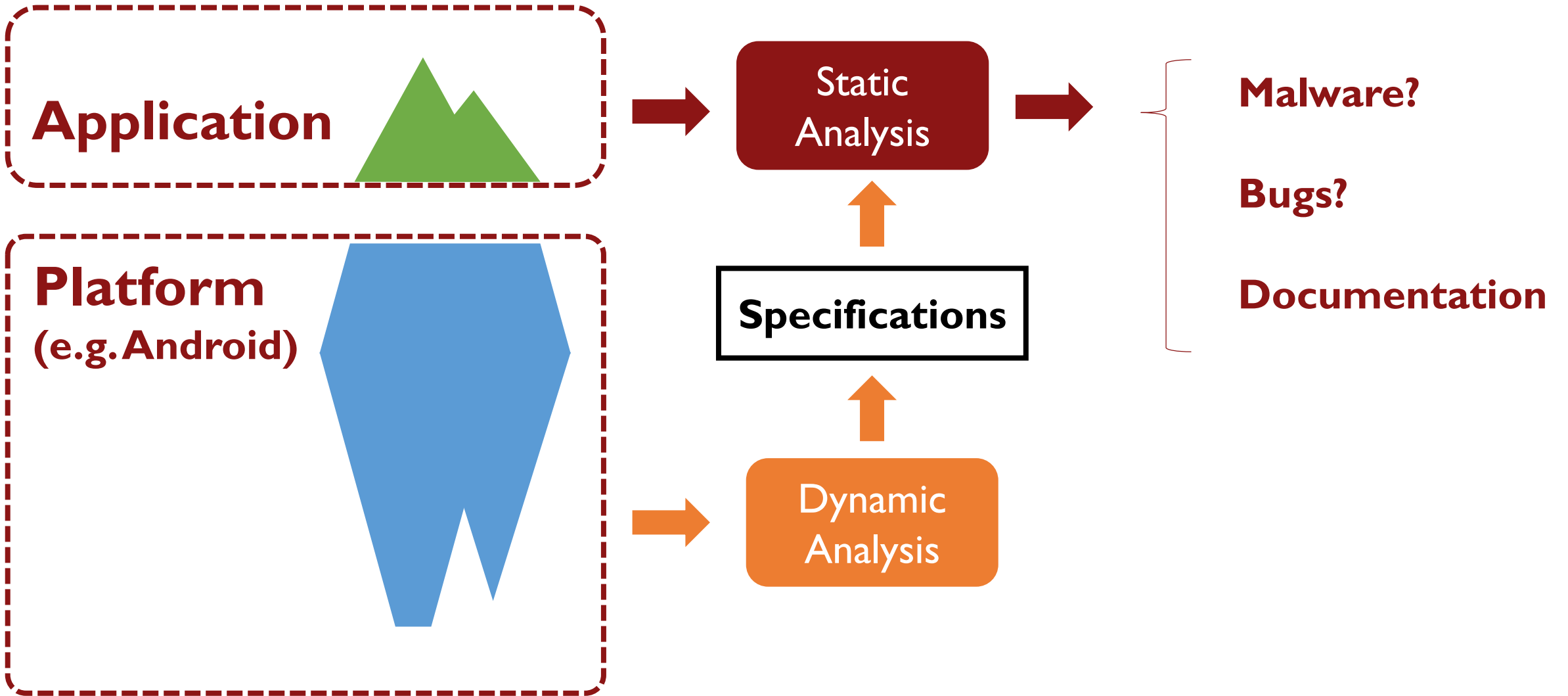


**Specifications**

Dynamic  
Analysis



# Mining Specifications





# Information flow specifications

# Static taint analysis



S.T.A.M.P.  
Static  
Analysis



## Information Flow Report

```
#LOCATION -> !  
INTERNET  
  
#CONTACTS -> !  
INTERNET  
  
#PHONE_NUM ->  
!INTERNET
```



Human  
Auditor



# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

*#PHONE\_NUM ->*

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

*#PHONE\_NUM -> ... -> ... -> ... -> !INTERNET*

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber, 0, 10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```



# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber, 0, 10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

```
TelephonyManager.getLine1Number()
    #PHONE_NUM -> return
```

*#PHONE\_NUM -> mPhoneNumber*

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

```
TelephonyManager.getLine1Number()
    #PHONE_NUM -> return
```

```
CharBuffer.put(String,int,int)
    arg#1 -> this
    arg#1 -> return
    this -> return
```

*#PHONE\_NUM -> mPhoneNumber -> b1*

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber,0,10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

```
TelephonyManager.getLine1Number()
    #PHONE_NUM -> return
```

```
CharBuffer.put(String,int,int)
    arg#1 -> this
    arg#1 -> return
    this -> return
```

```
CharsetEncoder.encode(CharBuffer)
    arg#1 -> return
```

*#PHONE\_NUM -> mPhoneNumber -> b1 -> bytebuffer*

# Information flow specifications

```
// Set-up
SocketChannel socket = ...;
CharBuffer buffer = ...;
CharsetEncoder encoder = ...;
TelephonyManager tMgr = ...;

// Leak phone number
// ( #PHONE_NUM -> !INTERNET )
String mPhoneNumber = tMgr.getLine1Number();
CharBuffer b1 = buffer.put(mPhoneNumber, 0, 10);
ByteBuffer bytebuffer = encoder.encode(b1);
socket.write(bytebuffer);
```

```
TelephonyManager.getLine1Number()
    #PHONE_NUM -> return
```

```
CharBuffer.put(String,int,int)
    arg#1 -> this
    arg#1 -> return
    this -> return
```

```
CharsetEncoder.encode(CharBuffer)
    arg#1 -> return
```

```
SocketChannel.write(ByteBuffer)
    arg#1 -> !INTERNET
```

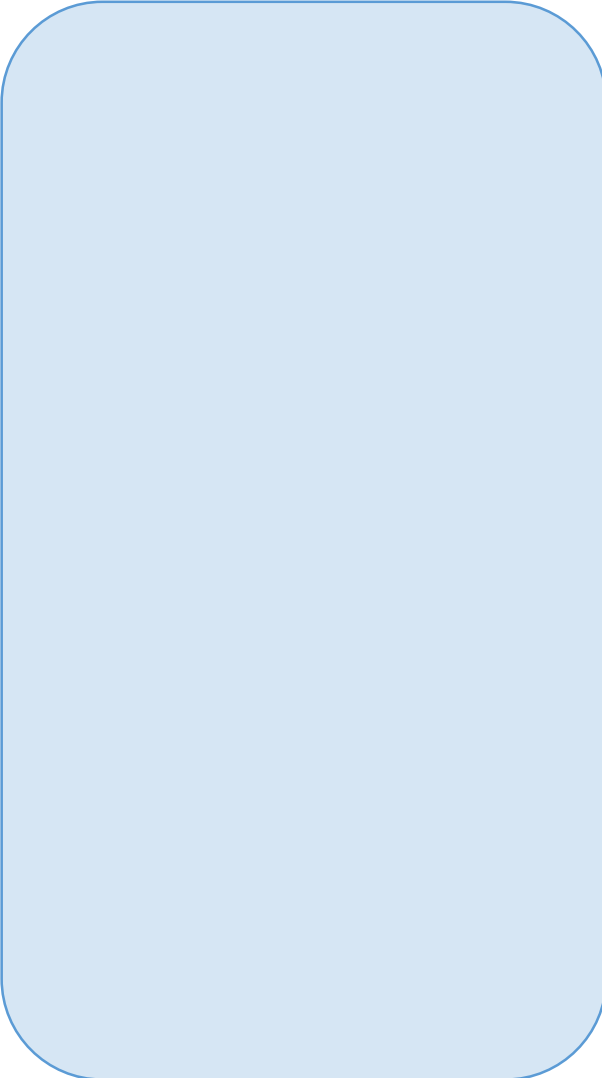
*#PHONE\_NUM -> mPhoneNumber -> b1 -> bytebuffer -> !INTERNET*



Technique

# Instrument, run, analyze

Instrument



Run

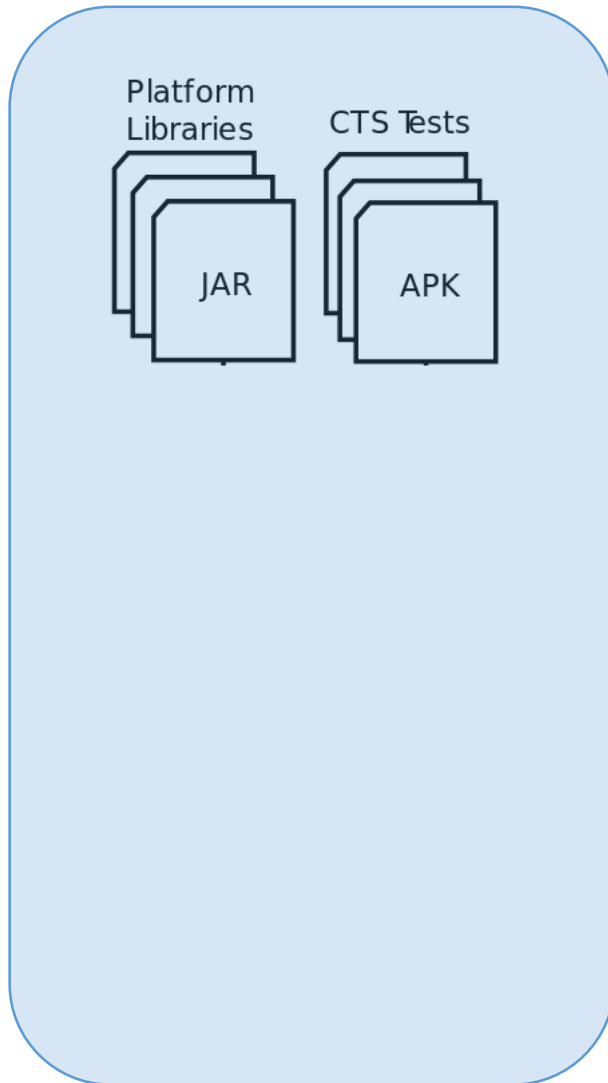


Analyze

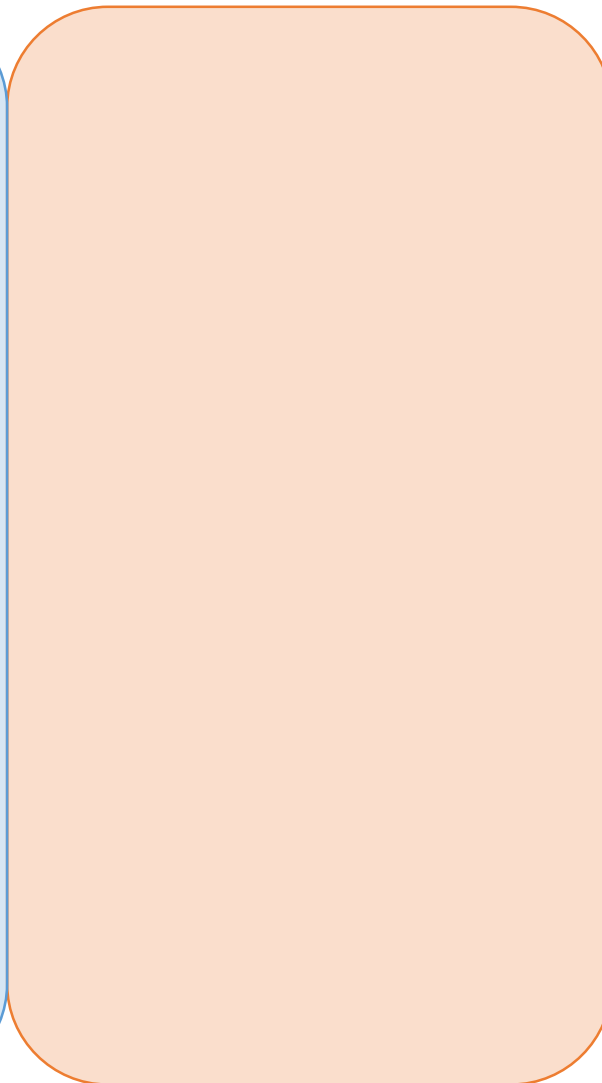


# Instrument, run, analyze

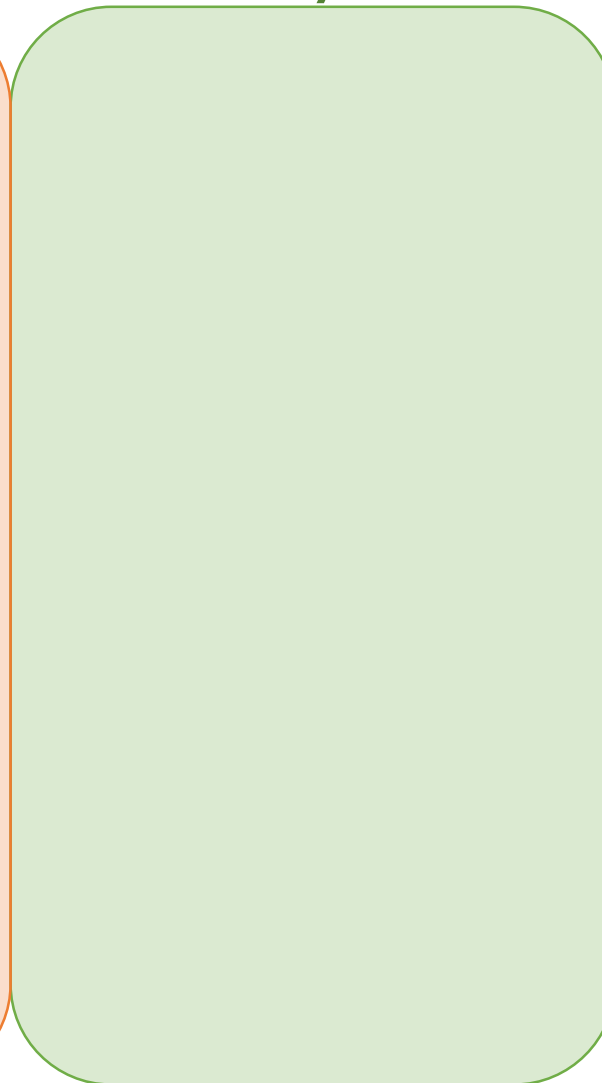
Instrument



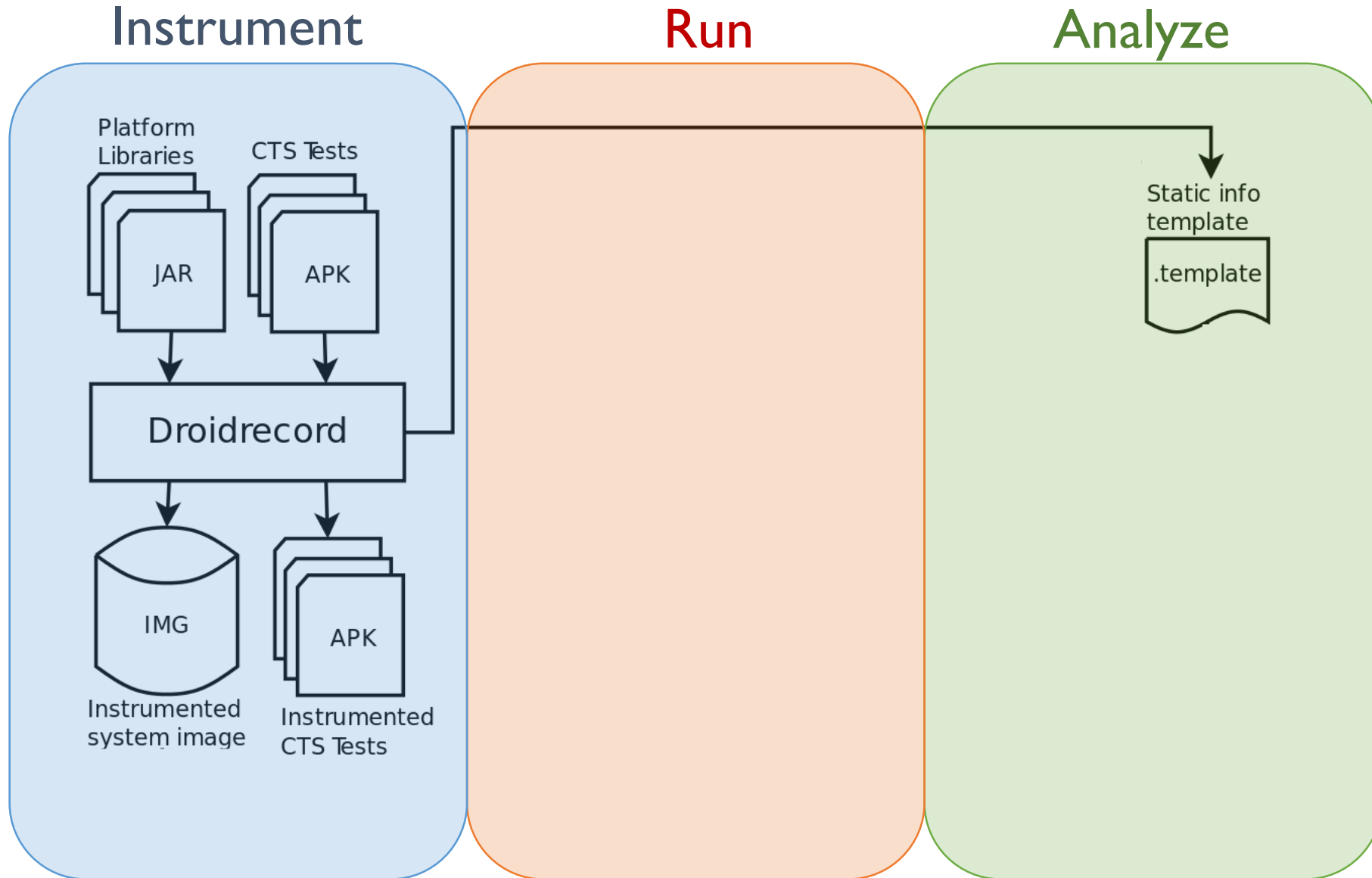
Run



Analyze

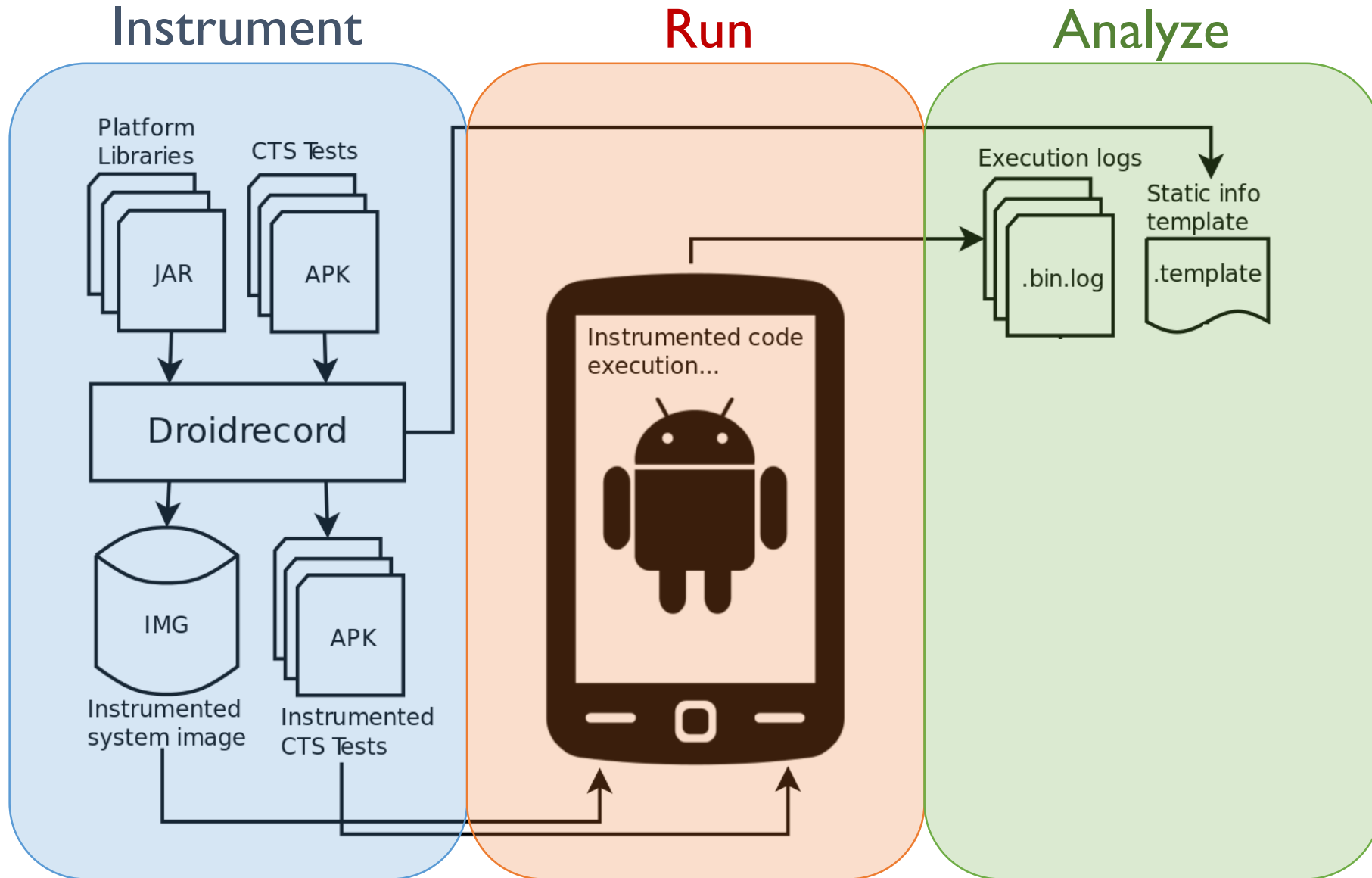


# Instrument, run, analyze

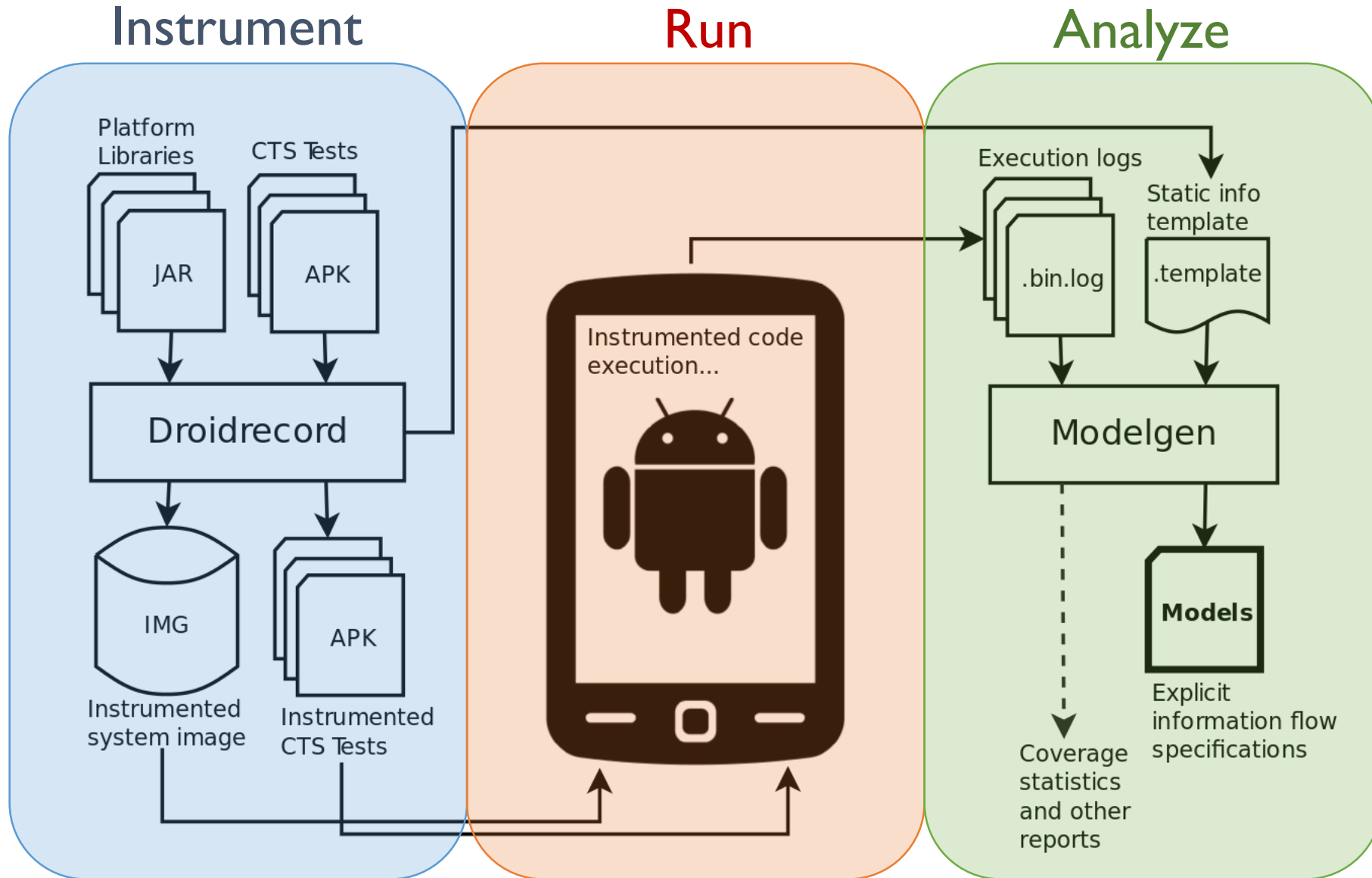




# Instrument, run, analyze



# Instrument, run, analyze



# Method trace

**Definition:**

# Method trace

## **Definition:**

- **Sequence of recorded operations between method entry and return.**

# Method trace

## **Definition:**

- **Sequence of recorded operations between method entry and return.**
- **Including calls to other methods.**

# Example

```
o . m ( arg1 , arg2 ) :  
    t = arg1 ⊗ arg2  
    o1 = o.f  
    o2 = o1.g  
    o3 = o.g  
    o2.f = t  
    return o
```

# Example

```
o . m ( arg1 , arg2 ) :  
    t = arg1 ⊗ arg2  
    o1 = o.f  
    o2 = o1.g  
    o3 = o.g  
    o2.f = t  
    return o
```

# Example

```
o . m ( arg1 , arg2 ) :  
    t = arg1 ⊗ arg2  
    o1 = o.f  
    o2 = o1.g  
    o3 = o.g  
    o2.f = t  
    return o
```



# Example

```
o . m ( arg1 , arg2 ) :  
    t = arg1 ⊗ arg2  
    o1 = o.f  
    o2 = o1.g  
    o3 = o.g  
    o2.f = t  
    return o
```

# Example

**o . m ( arg1 , arg2 ) :**

**t = arg1 ⊗ arg2**

**o1 = o.f**

**o2 = o1.g**

**o3 = o.g**

**o2.f = t**

**return o**

**Spec:**

arg1->this

arg2->this

# Example

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o.f  
  o2 = o1.g  
  o3 = o.g  
  o2.f = t  
  return o
```

**Spec:**

**arg1->this**

**arg2->this**

**this->return**

# Example

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o.f  
  o2 = o1.g  
  o3 = o.g  
  o2.f = t  
  return o
```

**Spec:**

arg1->this

arg2->this

this->return


arg1->return

arg2-> return

# Example: Initialization

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o . f  
  o2 = o1 . g  
  o3 = o . g  
  o2 . f = t  
  return o
```

```
ret = o . m ( arg1 , arg2 )
```



# Example: Taint propagation

**o . m ( arg1 , arg2 ) :**

**t = arg1 ⊗ arg2**

**o1 = o . f**

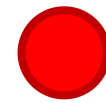
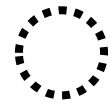
**o2 = o1 . g**

**o3 = o . g**

**o2 . f = t**

**return o**

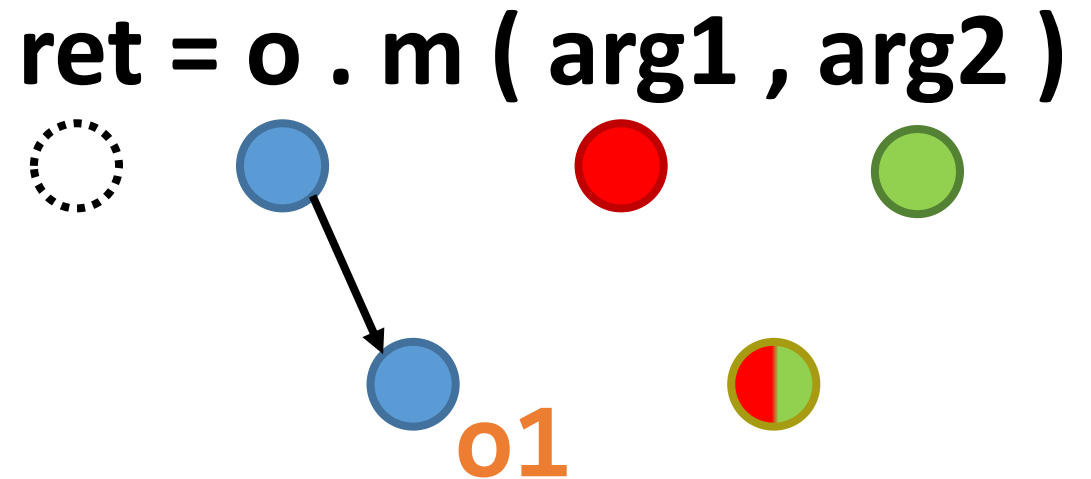
**ret = o . m ( arg1 , arg2 )**



**t**

# Example: Loads

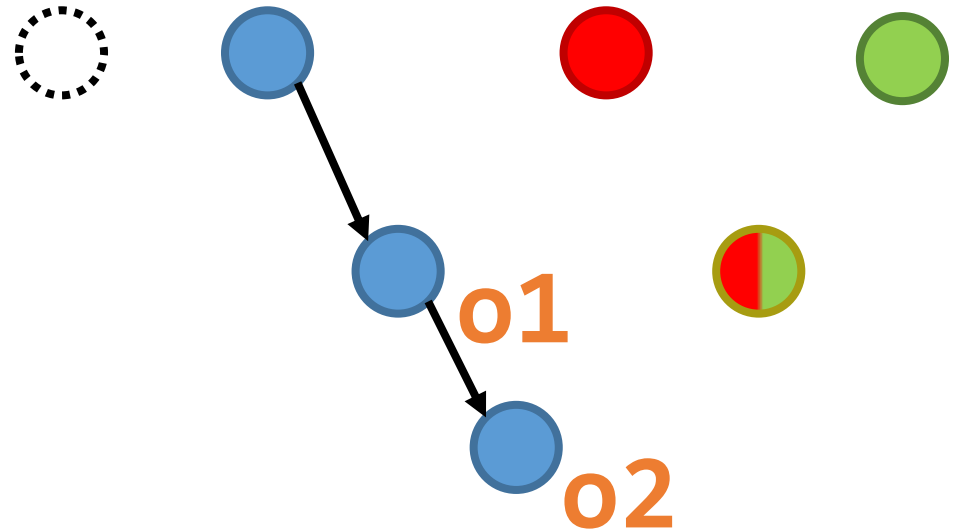
```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o.f  
  o2 = o1.g  
  o3 = o.g  
  o2.f = t  
  return o
```



# Example: Loads

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o.f  
  o2 = o1.g  
  o3 = o.g  
  o2.f = t  
  return o
```

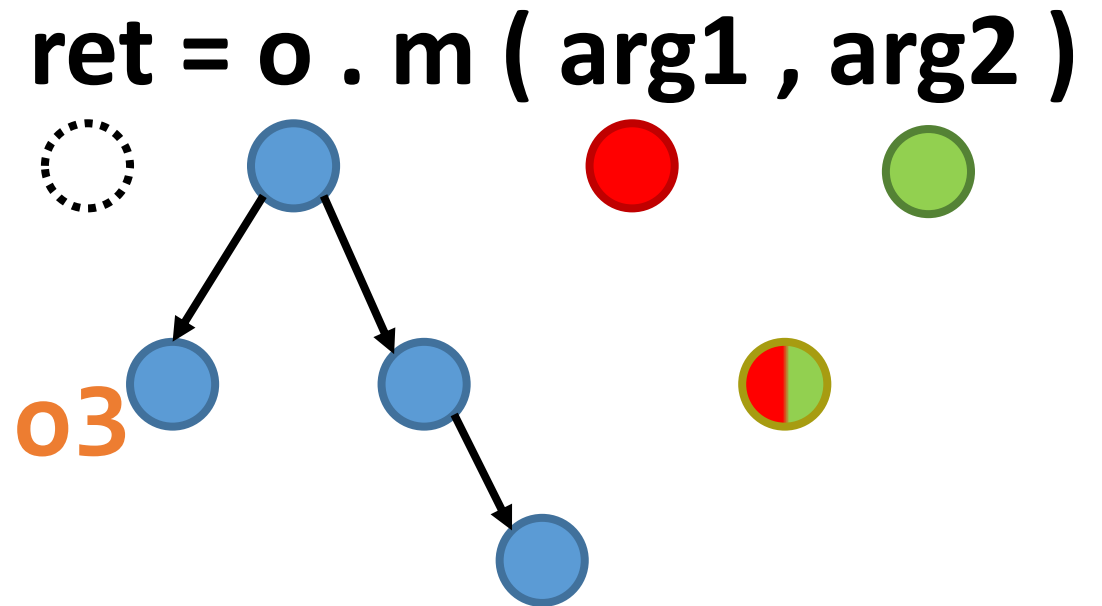
ret = o . m ( arg1 , arg2 )





# Example: Loads

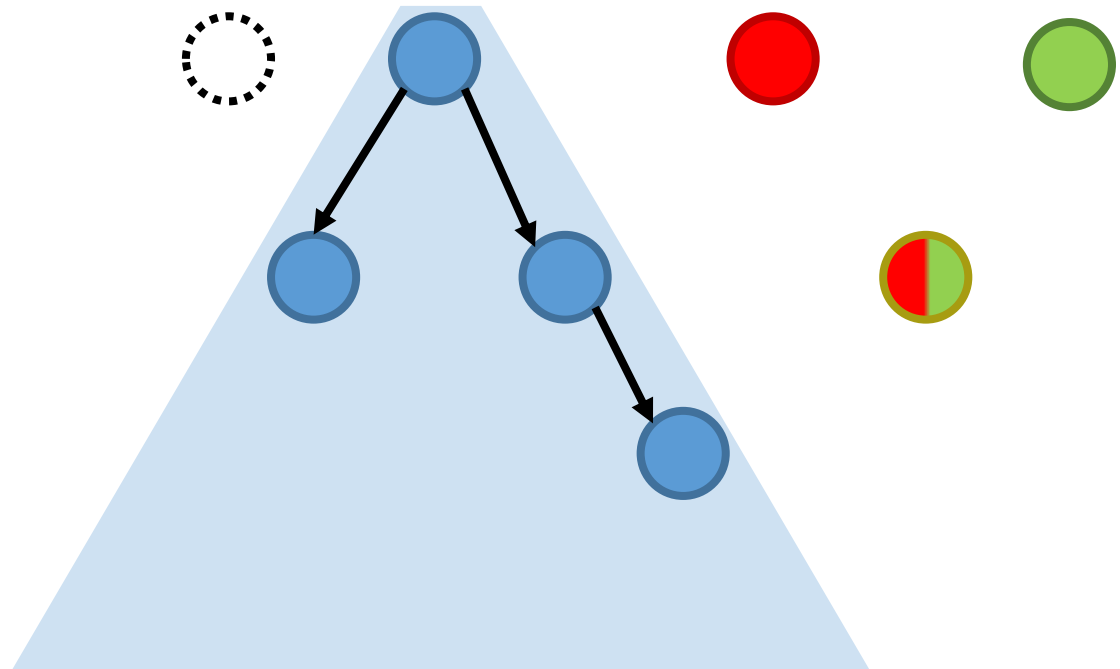
```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o . f  
  o2 = o1 . g  
  o3 = o . g  
  o2 . f = t  
  return o
```



# Example: Loads

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o . f  
  o2 = o1 . g  
  o3 = o . g  
  o2 . f = t  
  return o
```

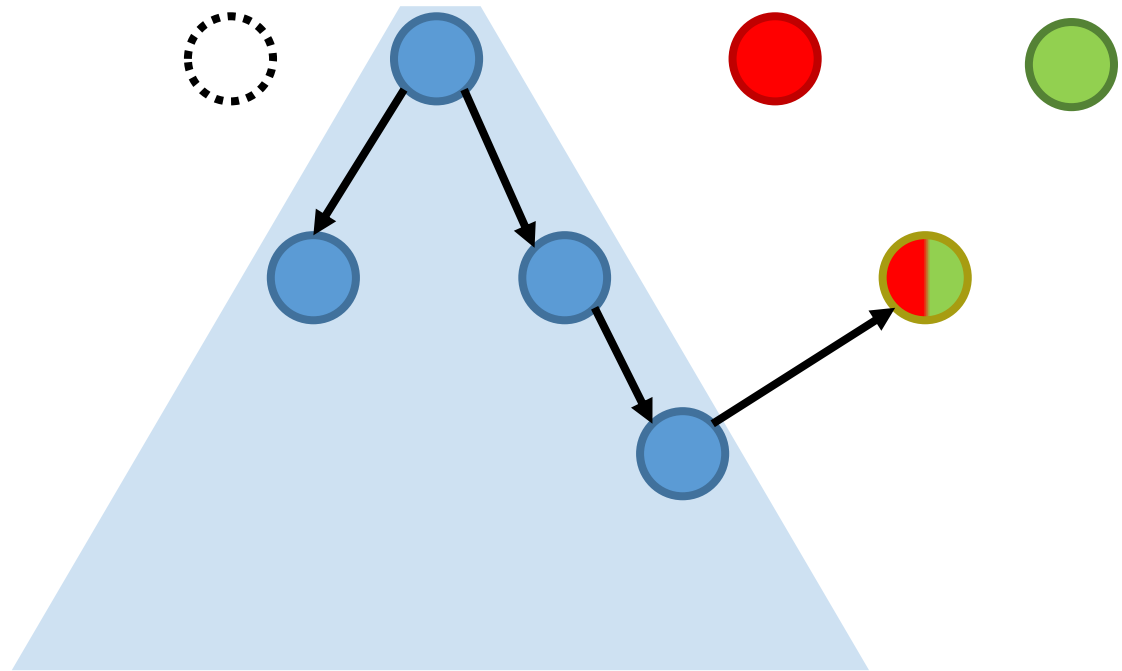
ret = o . m ( arg1 , arg2 )



# Example: Store

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o . f  
  o2 = o1 . g  
  o3 = o . g  
  o2.f = t  
  return o
```

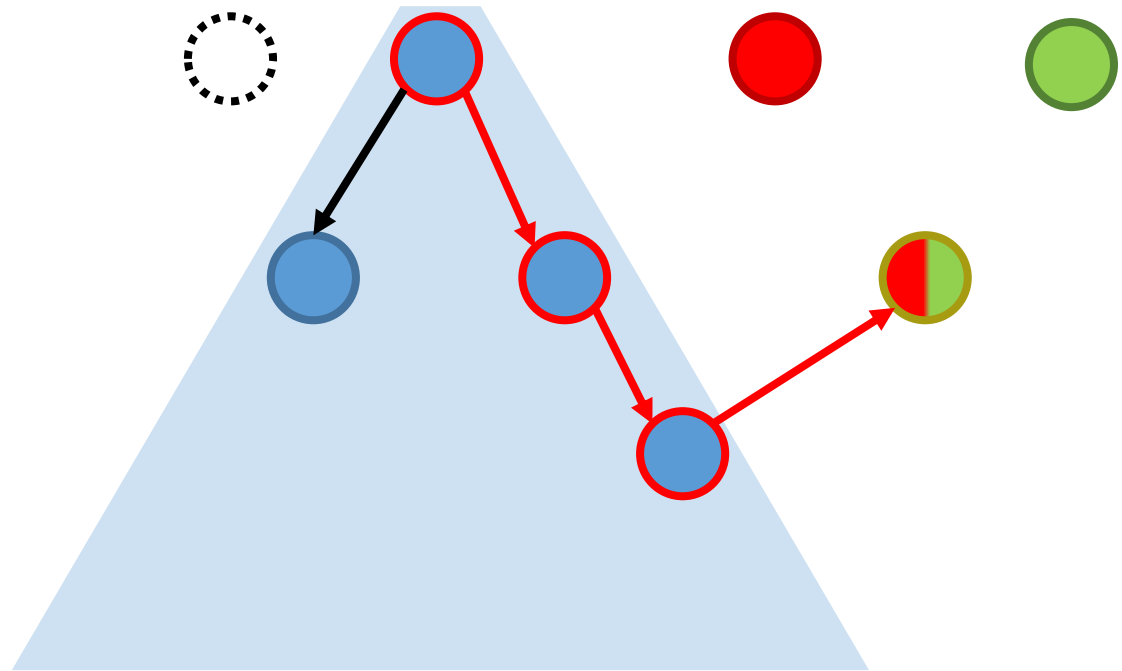
ret = o . m ( arg1 , arg2 )



# Example: Store

```
o . m ( arg1 , arg2 ) :  
  t = arg1 ⊗ arg2  
  o1 = o . f  
  o2 = o1 . g  
  o3 = o . g  
  o2.f = t  
  return o
```

ret = o . m ( arg1 , arg2 )



# Example: Store

$o.m(\text{arg1}, \text{arg2}) :$

$t = \text{arg1} \otimes \text{arg2}$

$o1 = o.f$

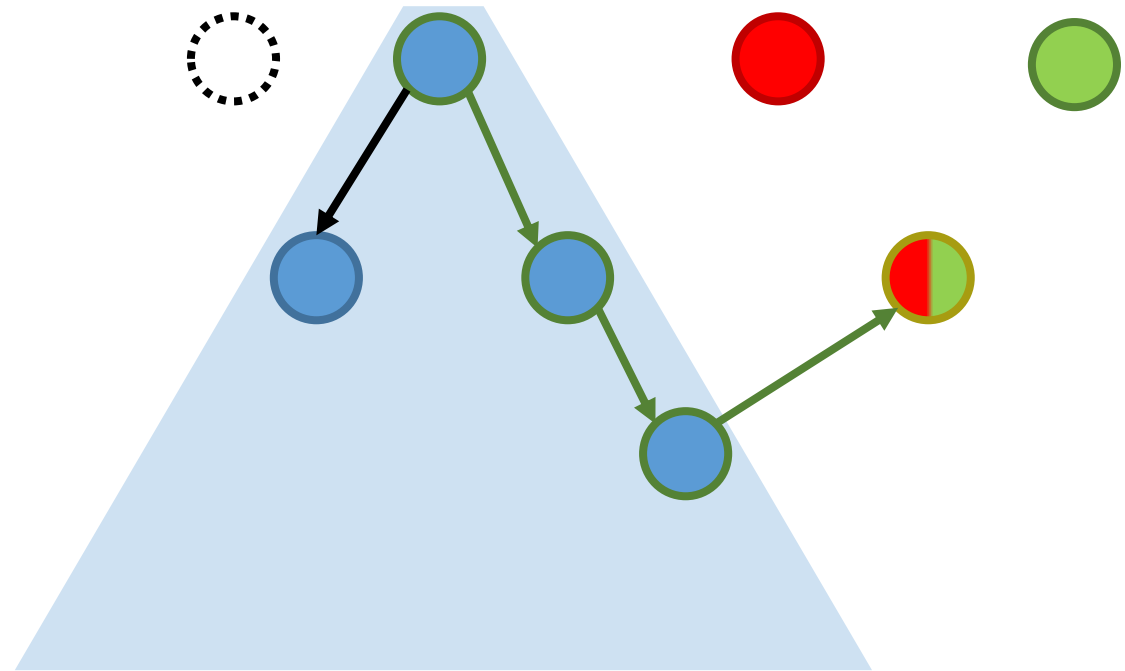
$o2 = o1.g$

$o3 = o.g$

$o2.f = t$

return  $o$

$\text{ret} = o.m(\text{arg1}, \text{arg2})$



# Example: Store

**`o . m ( arg1 , arg2 ) :`**

`t = arg1  $\otimes$  arg2`

`o1 = o.f`

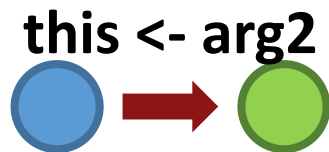
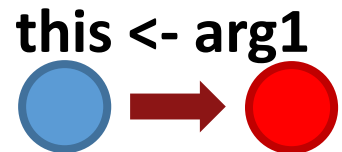
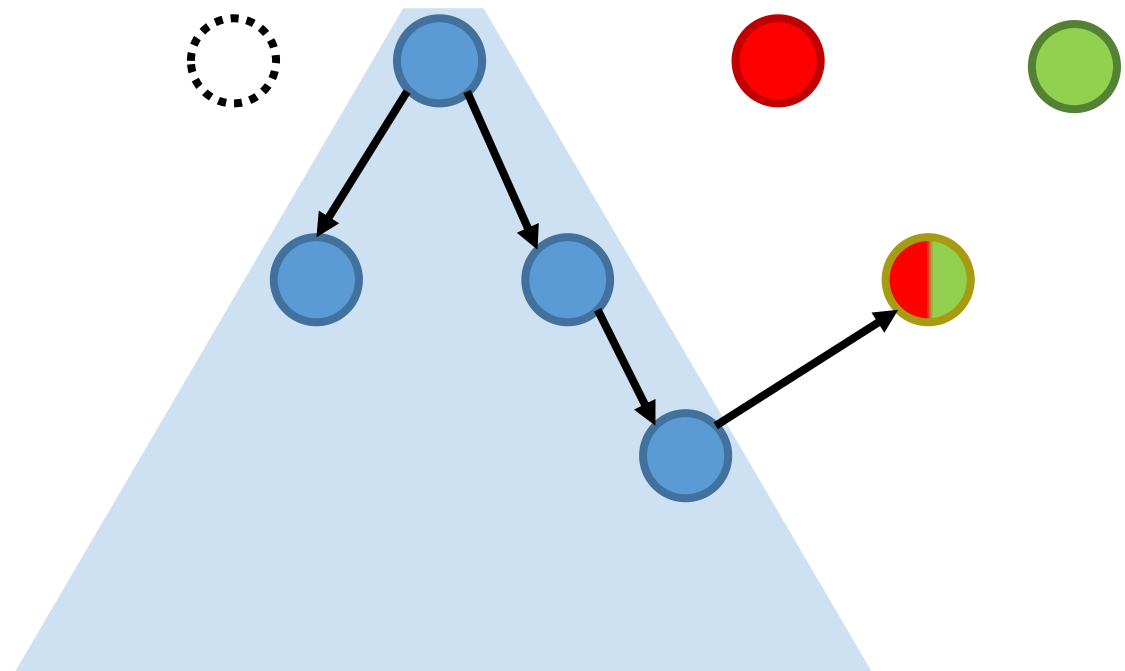
`o2 = o1.g`

`o3 = o.g`

`o2.f = t`

`return o`

**`ret = o . m ( arg1 , arg2 )`**



# Example: Store

**`o . m ( arg1 , arg2 ) :`**

`t = arg1  $\otimes$  arg2`

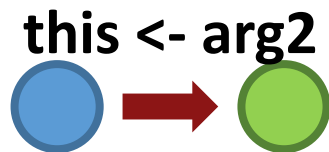
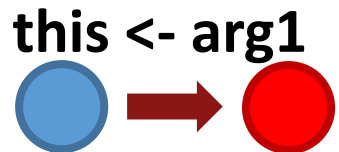
`o1 = o.f`

`o2 = o1.g`

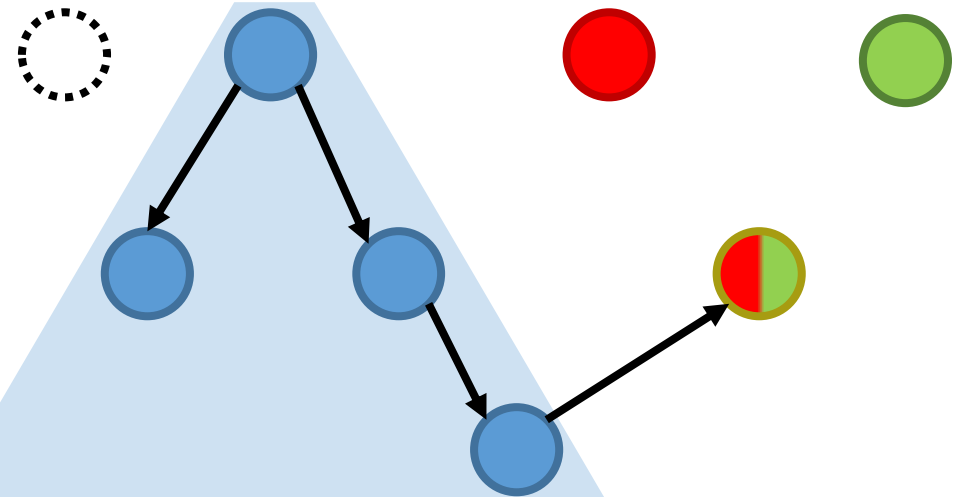
`o3 = o.g`

`o2.f = t`

`return o`



**`ret = o . m ( arg1 , arg2 )`**



**! : Information flow goes in the opposite direction of reachability**

# Example: Store

**`o . m ( arg1 , arg2 ) :`**

`t = arg1  $\otimes$  arg2`

`o1 = o.f`

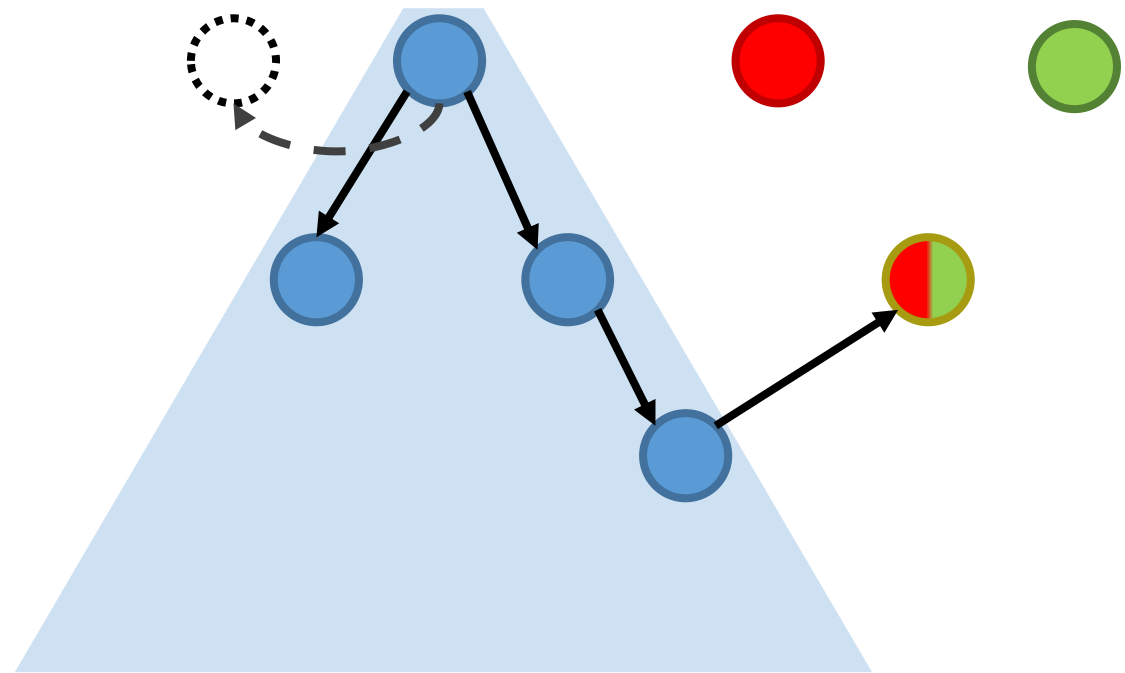
`o2 = o1.g`

`o3 = o.g`

`o2.f = t`

**return o**

**`ret = o . m ( arg1 , arg2 )`**



`this <- arg1`



`this <- arg2`





# Example: Store

**`o . m ( arg1 , arg2 ) :`**

`t = arg1  $\otimes$  arg2`

`o1 = o.f`

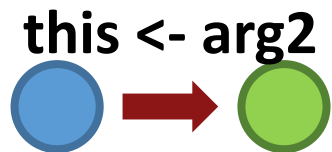
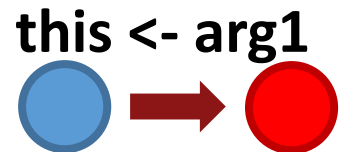
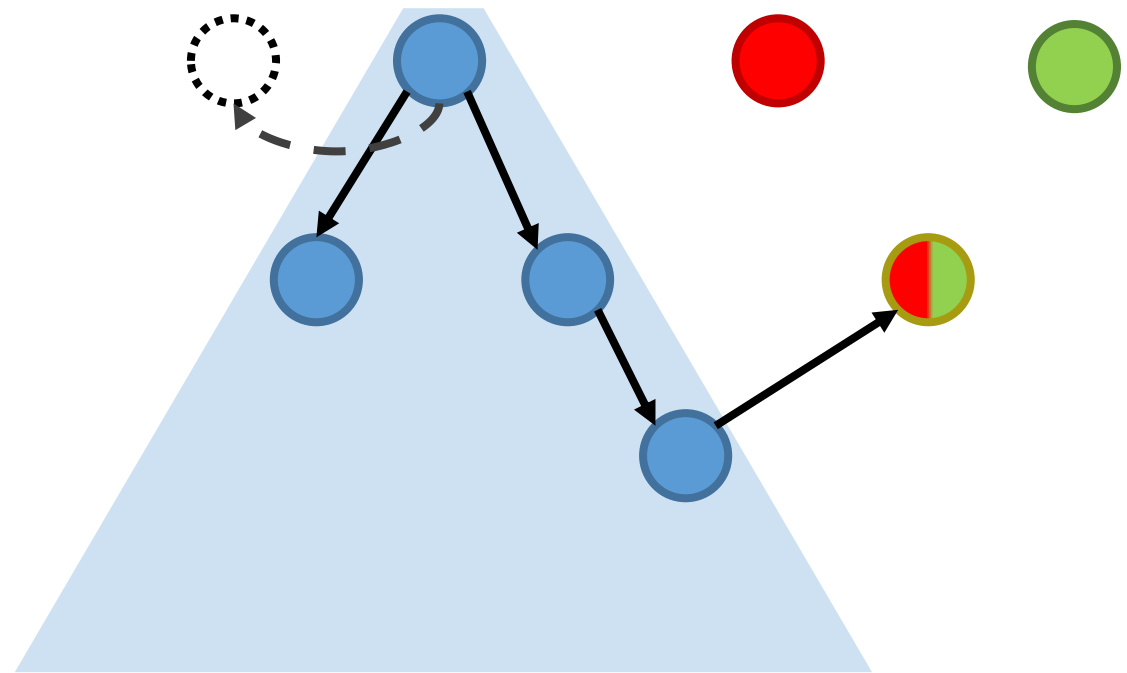
`o2 = o1.g`

`o3 = o.g`

`o2.f = t`

**return o**

**`ret = o . m ( arg1 , arg2 )`**



# Example: Store

**`o . m ( arg1 , arg2 ) :`**

`t = arg1  $\otimes$  arg2`

`o1 = o.f`

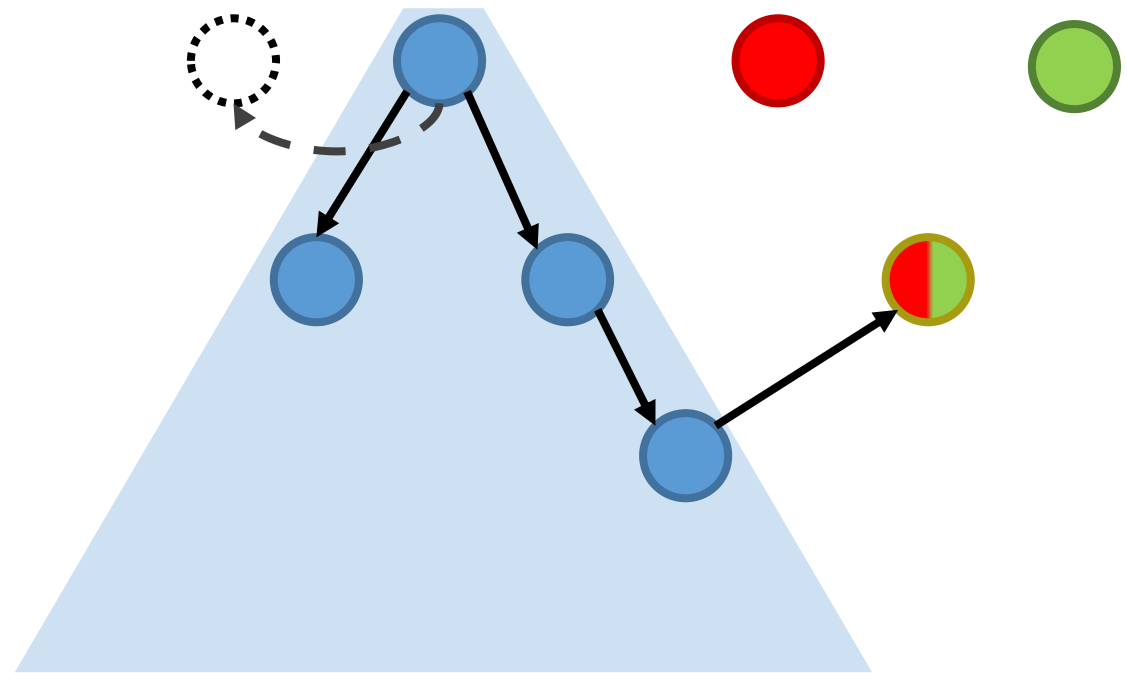
`o2 = o1.g`

`o3 = o.g`

`o2.f = t`

**return o**

**`ret = o . m ( arg1 , arg2 )`**



`this <- arg1`



`this <- arg2`



`return <- this`



`return <- arg1`



`return <- arg2`



# Example: Store

## Spec:

arg1->this

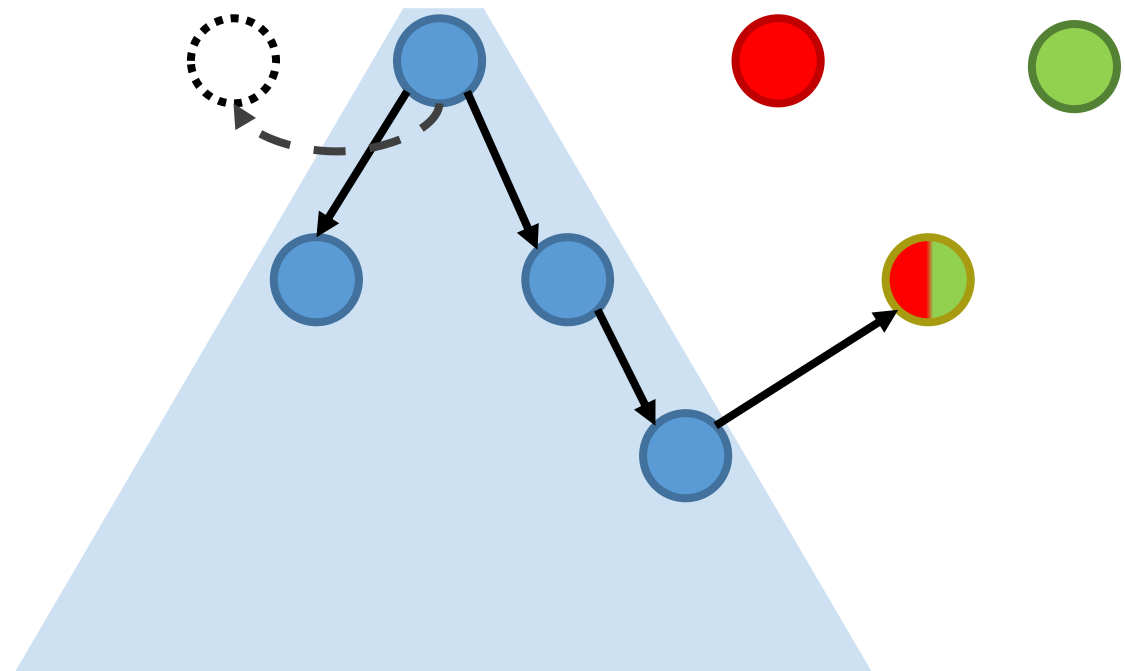
arg2->this

this->return

arg1->return

arg2-> return

ret = o . m ( arg1 , arg2 )



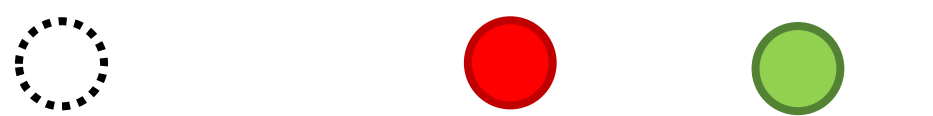
# Merging specifications


$r = \max(\text{arg1}, \text{arg2})$



# Merging specifications




$r = \max ( 5 , 3 )$









  
return <- arg1

Trace 1




# Merging specifications

**r = max ( 5 , 3 )**  
  

**r = max ( 2 , 7 )**  
  

    
return <- arg1

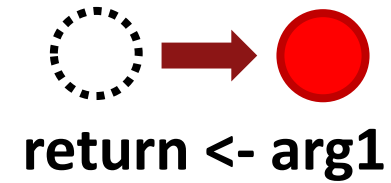
Trace I

    
return <- arg2

Trace II

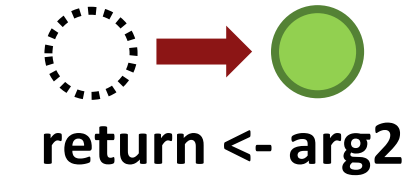
# Merging specifications

$r = \max ( 5 , 3 )$



Trace I

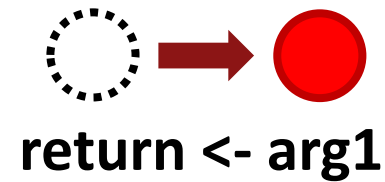
$r = \max ( 2 , 7 )$



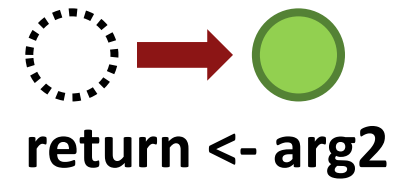
Trace II

---

$r = \max ( \text{arg1} , \text{arg2} )$



U



# Notes and gotchas

- **Native code / instrumentation holes**
- **Arrays, threading, exceptions**
- **Method calls (and recursion)**
- **Etc.**



# Notes and gotchas

- Native code instrumentation holes
- Array reads
- Method calls (
- Etc.

See paper

## Modelgen: Mining Explicit Information Flow Specifications from Concrete Executions



Lazaro Clapp  
Stanford University, USA  
lazaro@stanford.edu

Saswat Anand  
Stanford University, USA  
saswat@cs.stanford.edu

Alex Aiken  
Stanford University, USA  
aiken@cs.stanford.edu

### ABSTRACT

We present a technique to mine explicit information flow specifications from concrete executions. These specifications can be consumed by a static taint analysis, enabling static analysis to work even when method definitions are missing or portions of the program are too difficult to analyze statically (e.g., due to dynamic features such as reflection). We present an implementation of our technique for the Android platform. When compared to a set of manually written spec-

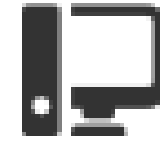
of the framework. However, there are at least four problems that make the analysis of framework code challenging. First, a very precise analysis of a framework may not scale because most frameworks are very large. Second, framework code may use dynamic language features, such as reflection in Java, which are difficult to analyze statically. Third, frameworks typically use non-code artifacts (e.g., configuration files) that have special semantics that must be modeled for accurate results. Fourth, frameworks usually build on ab-

# IV

## Experiments and results

# Experiment I: Man vs Machine

**309 methods, 51 classes**

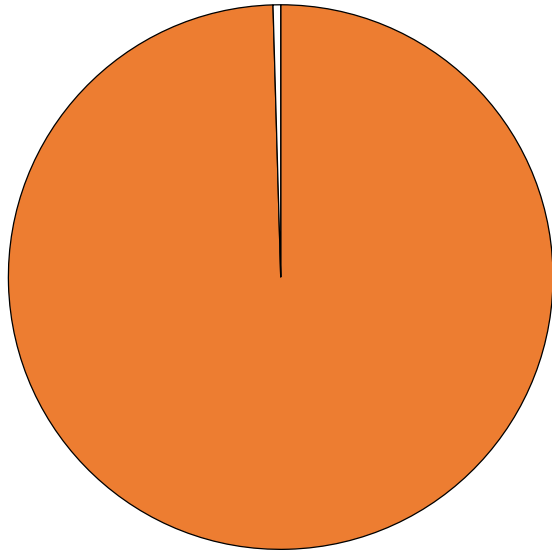


# Experiment I: Man vs Machine

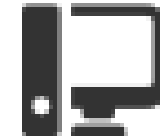
**309 methods, 51 classes**



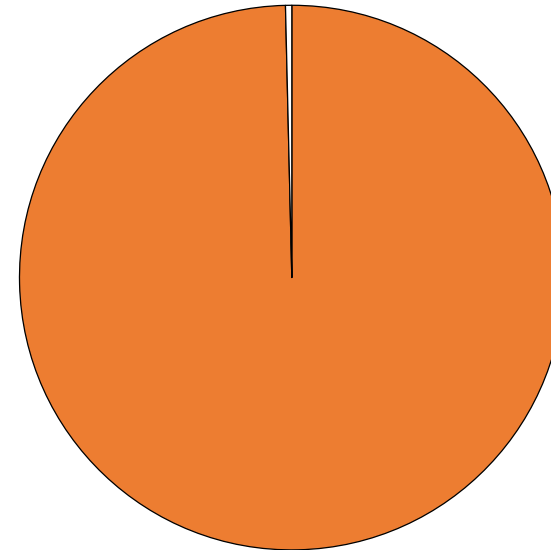
**99.55% Precision**



**440 TP / 2 FP**



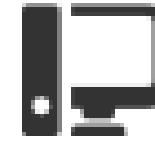
**99.63% Precision**



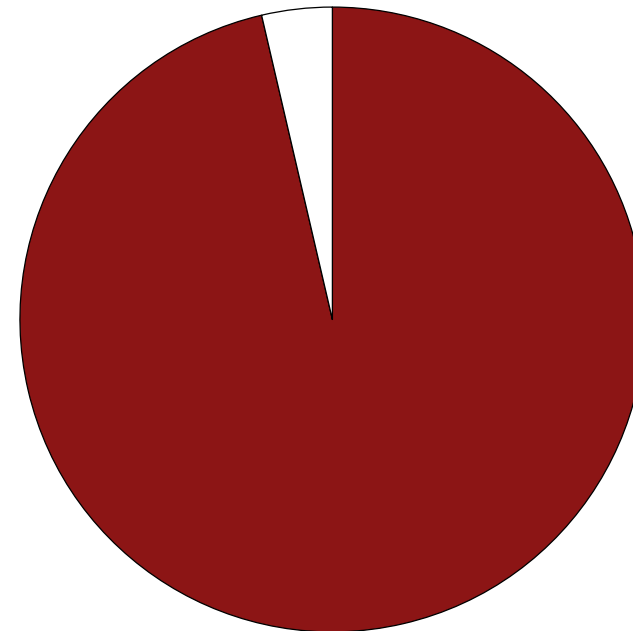
**540 TP / 2 FP**

# Experiment I: Man vs Machine

**309 methods, 51 classes**



**96.36% Recall vs Manual**

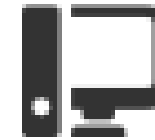
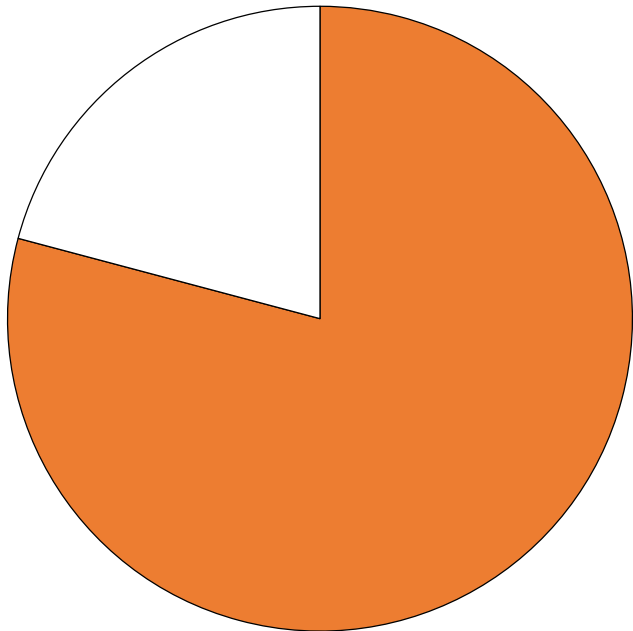


# Experiment I: Man vs Machine

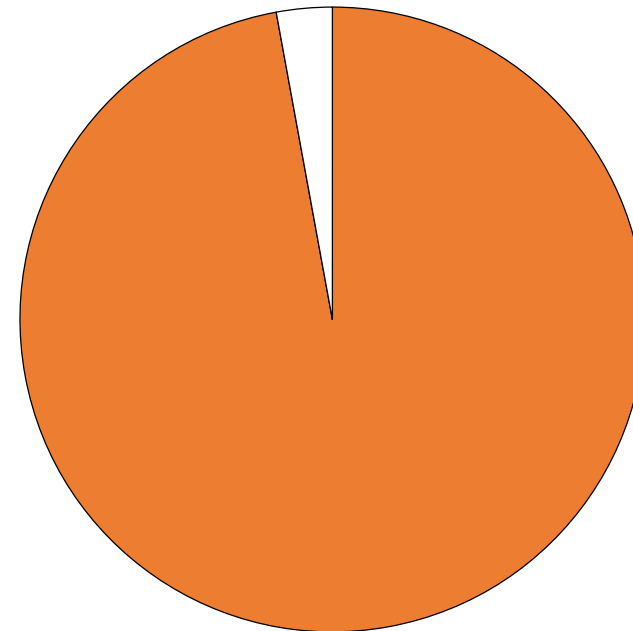
**309 methods, 51 classes**



**79.14% Recall vs Total (TP)**



**97.12% Recall vs Total (TP)**



# Experiment II: STAMP

# Experiment II: STAMP

- **242 apps (Google Play)**
- **Base: 3.08 flows (x app)**
- **Modelgen: 4.07 flows (x app)**



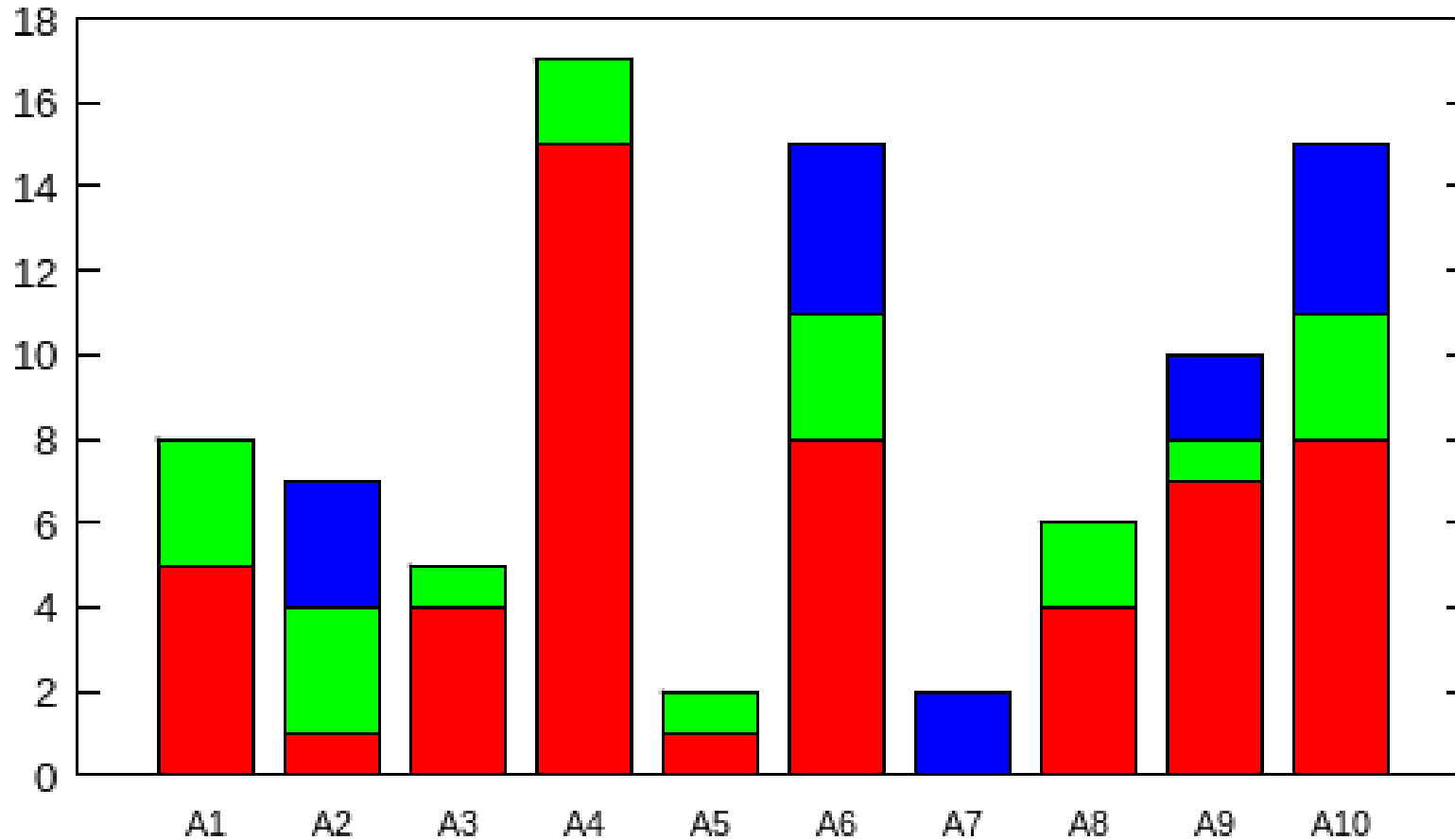
# Experiment II: STAMP

- **242 apps (Google Play)**
- **Base:** 1.07 flows (x app)
- **Modelgen:** 4.07 flows (x app)

But, are this... true positives?

# Experiment II: STAMP

**Flows**

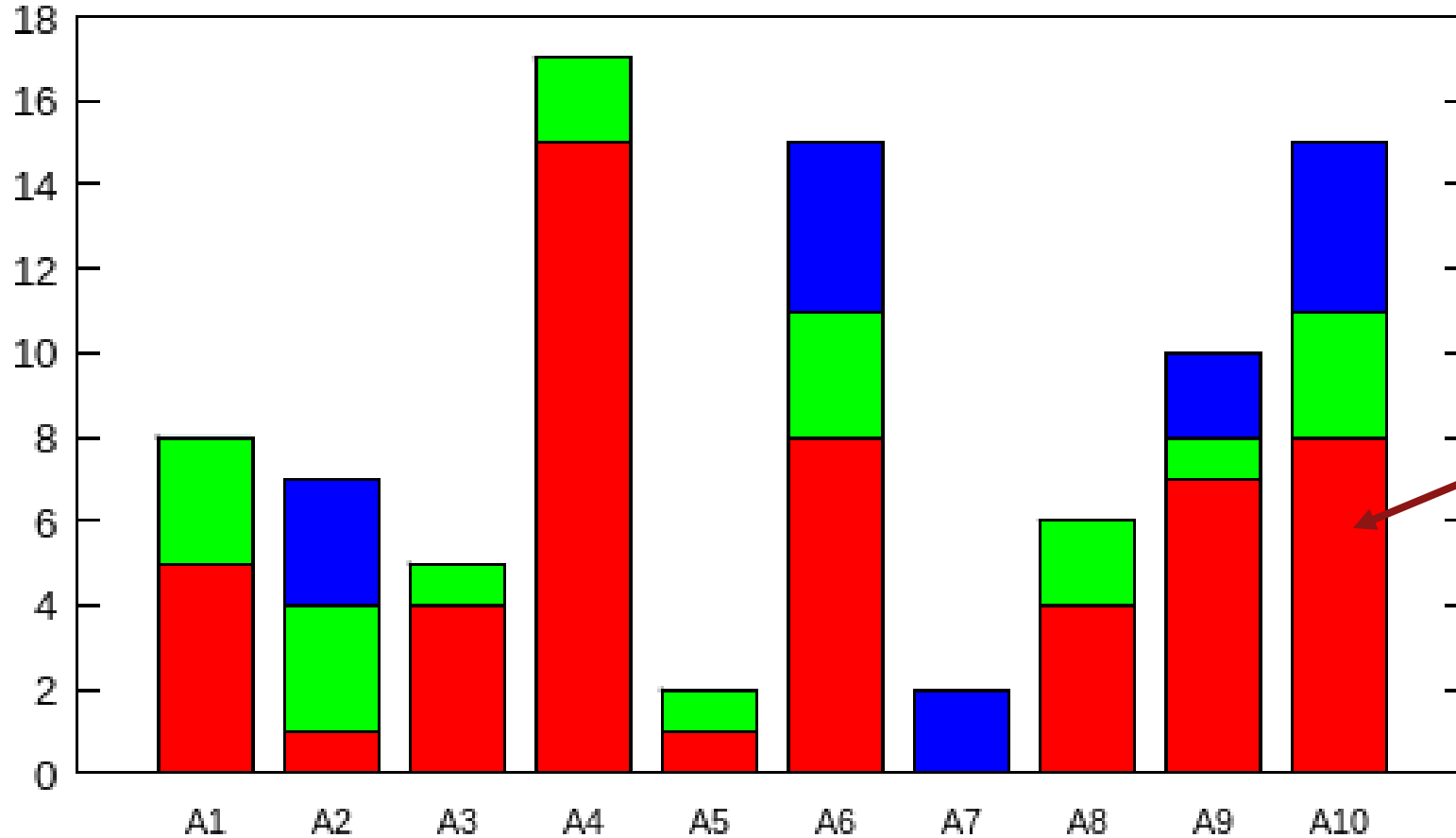


**Apps**

- Augmented configuration: Unknown
- Augmented configuration: Probable True Positives
- Both configurations

# Experiment II: STAMP

**Flows**

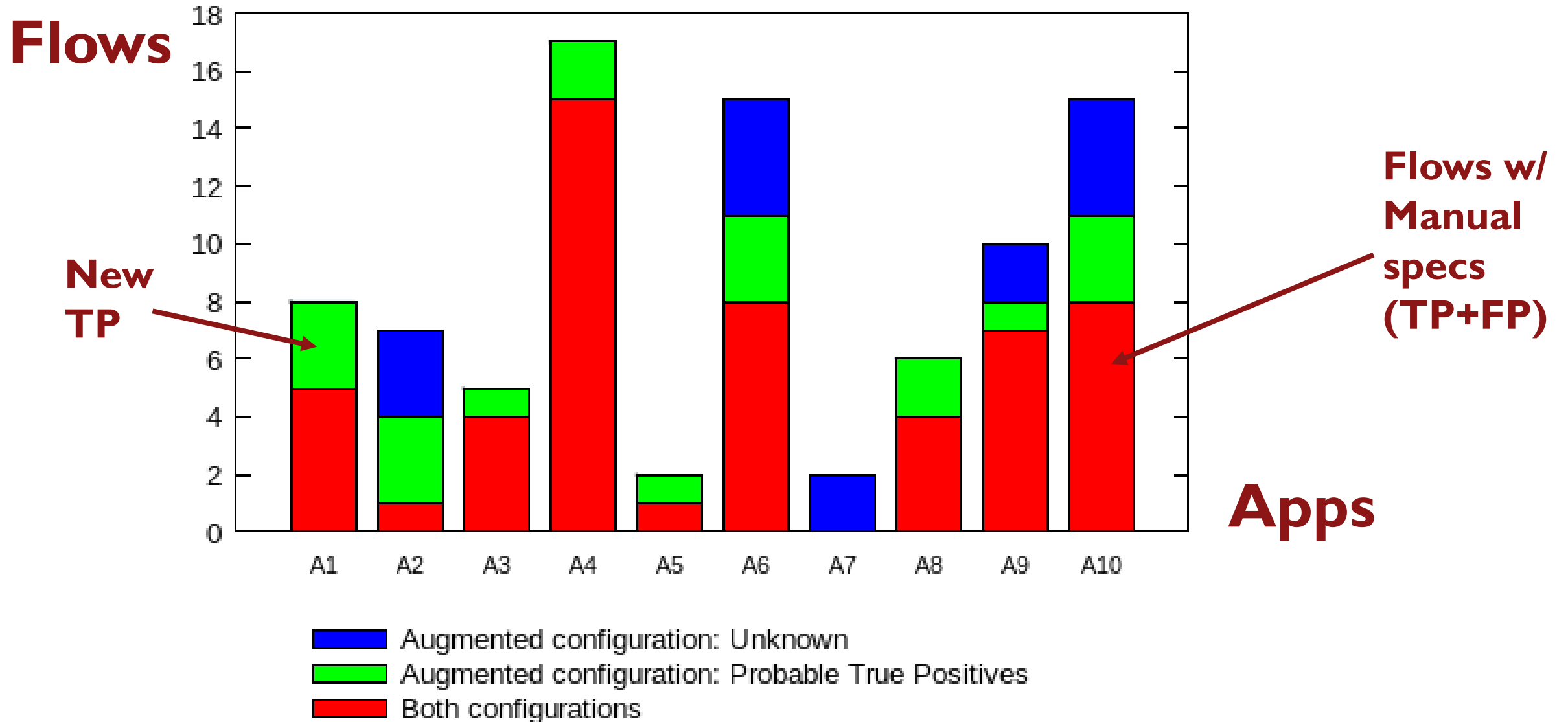


**Flows w/  
Manual  
specs  
(TP+FP)**

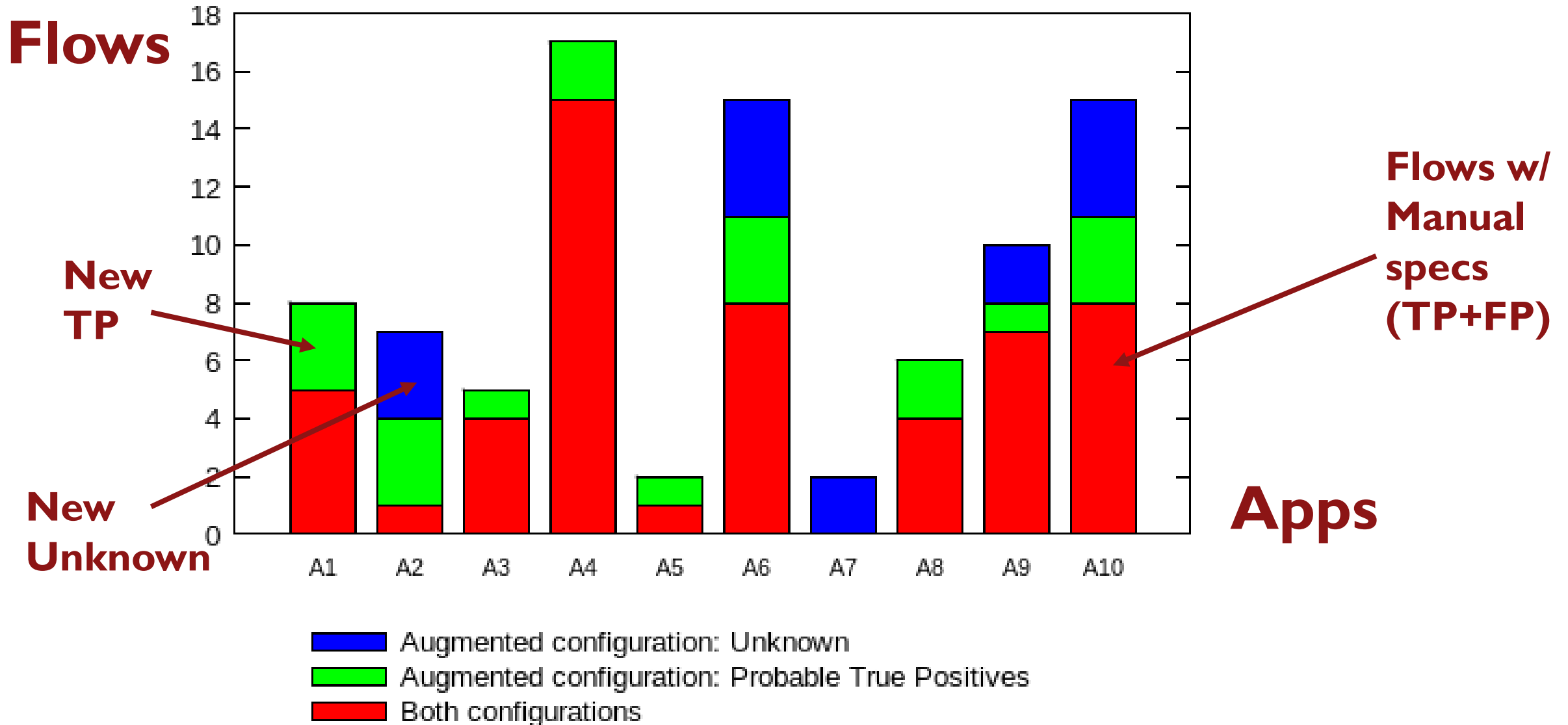
**Apps**

- Augmented configuration: Unknown
- Augmented configuration: Probable True Positives
- Both configurations

# Experiment II: STAMP



# Experiment II: STAMP



V

Conclusions and related work

# Key points

- **Platform code specifications**

# Key points

- **Platform code specifications**
- **Dynamic analysis > manual effort (sometimes)**



# Key points

- **Platform code specifications**
- **Dynamic analysis > manual effort (sometimes)**
- **For IF Specs: > 97% precision and recall**

# (Some) Related work

## Dynamic techniques for generating API specifications

- V. K. Palepu, G. H. Xu, and J. A. Jones. Improving efficiency of dynamic analysis with dynamic dependence summaries. ASE 2013
- A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. IEEE ToC, 1972.
- G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. POPL 2002.
- T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. ICSE 2006
- D. Lorenzoli, L. Mariani, and M. Pezze. Automatic generation of software behavioral models. ICSE 2008
- J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. ISSTA 2002

## Dynamic / Static taint analysis

- J. A. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. ISSTA 2007
- W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. OSDI 2010
- S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. PLDI 2014
- M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. OOPSLA 2011
- O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. POPL 2015

# Code and models available

[https://bitbucket.org/lazaro\\_clapp/droidrecord](https://bitbucket.org/lazaro_clapp/droidrecord)



Code and models available

[https://bitbucket.org/lazaro\\_clapp/droidrecord](https://bitbucket.org/lazaro_clapp/droidrecord)



Questions?