Security Analysis of LLVM Bitcode Files for Mobile Platforms

Vivek Sarkar

(vsarkar@rice.edu)

Department of Computer Science

Rice University

May 7, 2013

Acknowledgments

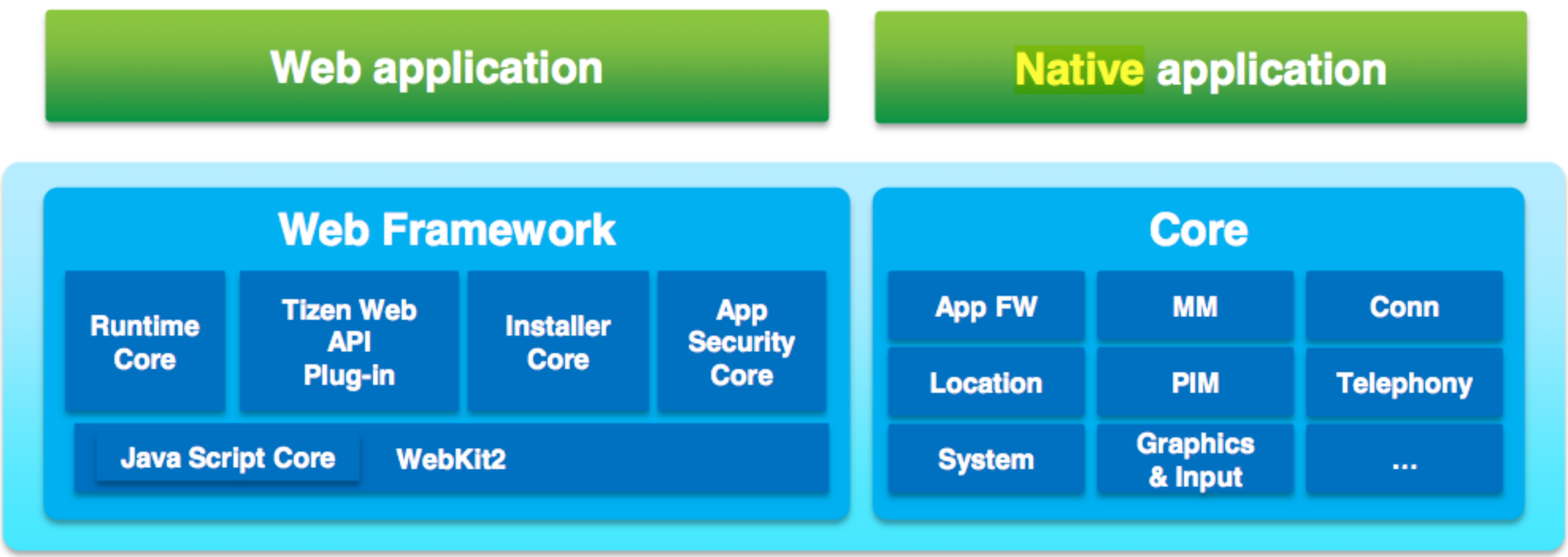
- Research contract from Samsung Electronics Co., Ltd. (Software R&D Center) for security analysis of Tizen applications
- Core team at Rice
  - Vivek Sarkar (PI)
  - Dan Wallach (Co-PI)
  - Michael Burke (Senior Research Scientist)
  - Jisheng Zhao (Research Scientist)
  - Deepak Majeti (PhD student)
  - Dragos Sbirlea (PhD student)
  - Bhargava Shastry (PhD student)
- Additional contributors at Rice
  - Keith Cooper
  - Swarat Chaudhuri
  - Additional PhD students
- Sources of support for other LLVM work
  - DARPA AACE program (2009 – 2012)
  - Support by Department of Defense and the National Science Foundation under a supplement to Grant No. 0964520 for parallel extensions to LLVM IR

Web vs. Native Applications on Mobile Platforms

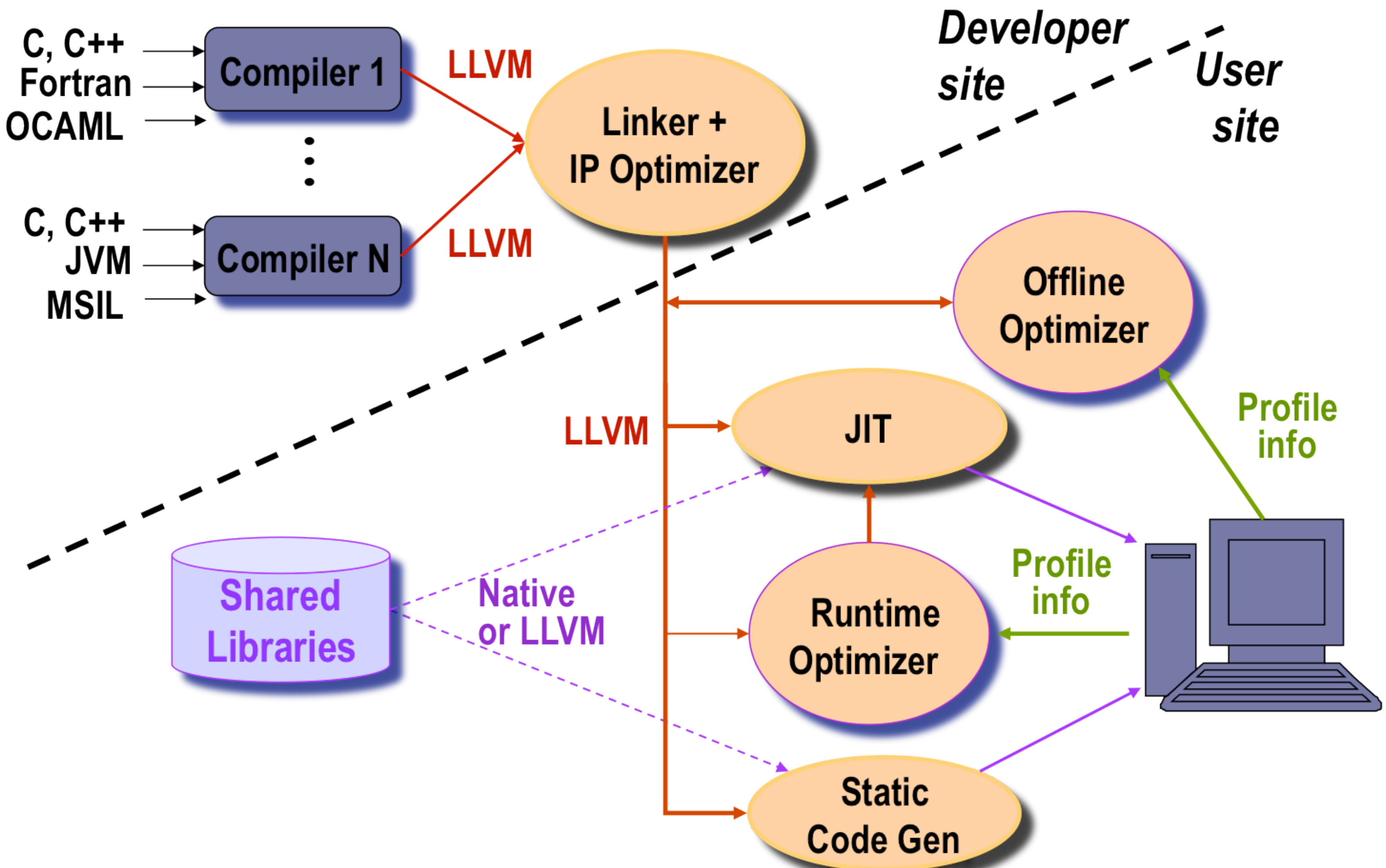
Example: Tizen open source platform (tizen.org)

- Web Framework is the primary application development environment
- Native Framework necessary for device-specific and performance-sensitive applications

(Source: “Tizen Overview and Architecture”, Seokjae Jeong, Samsung Electronics, Oct 2012)



LLVM vision: a Low-Level Virtual Machine



Source: CGO 2004 tutorial on LLVM, Chris Lattner & Vikram Adve

Benefits of using LLVM bitcode as distribution format for native applications

1. Retargetability: App Store can generate machine code for different devices
2. Safety: LLVM bitcode is amenable to security analysis

→ Focus of this talk

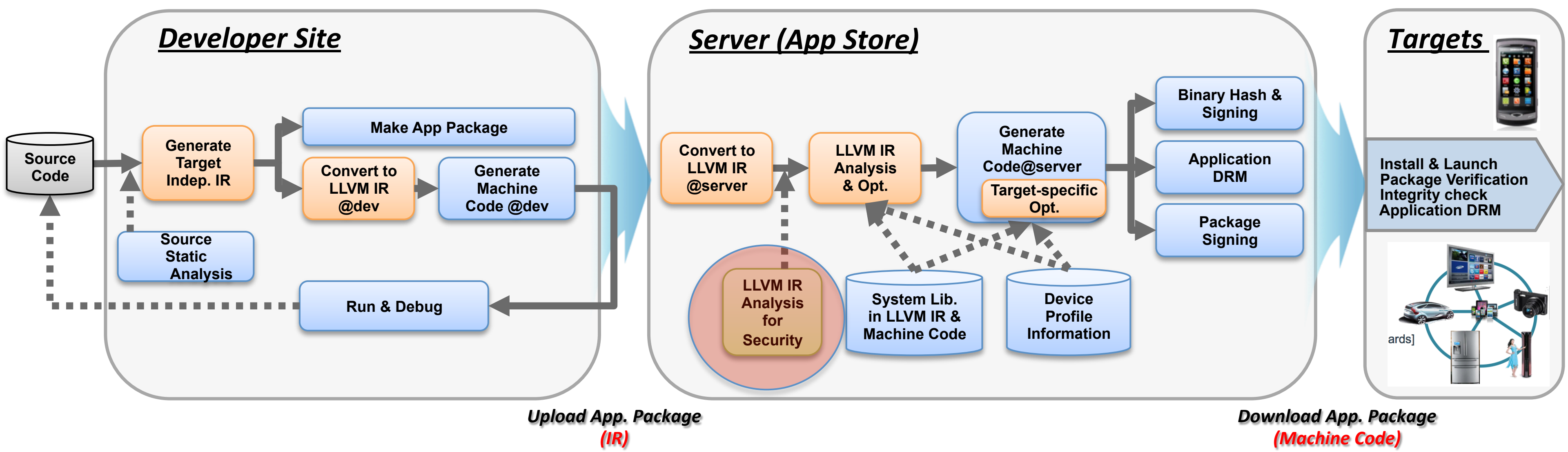
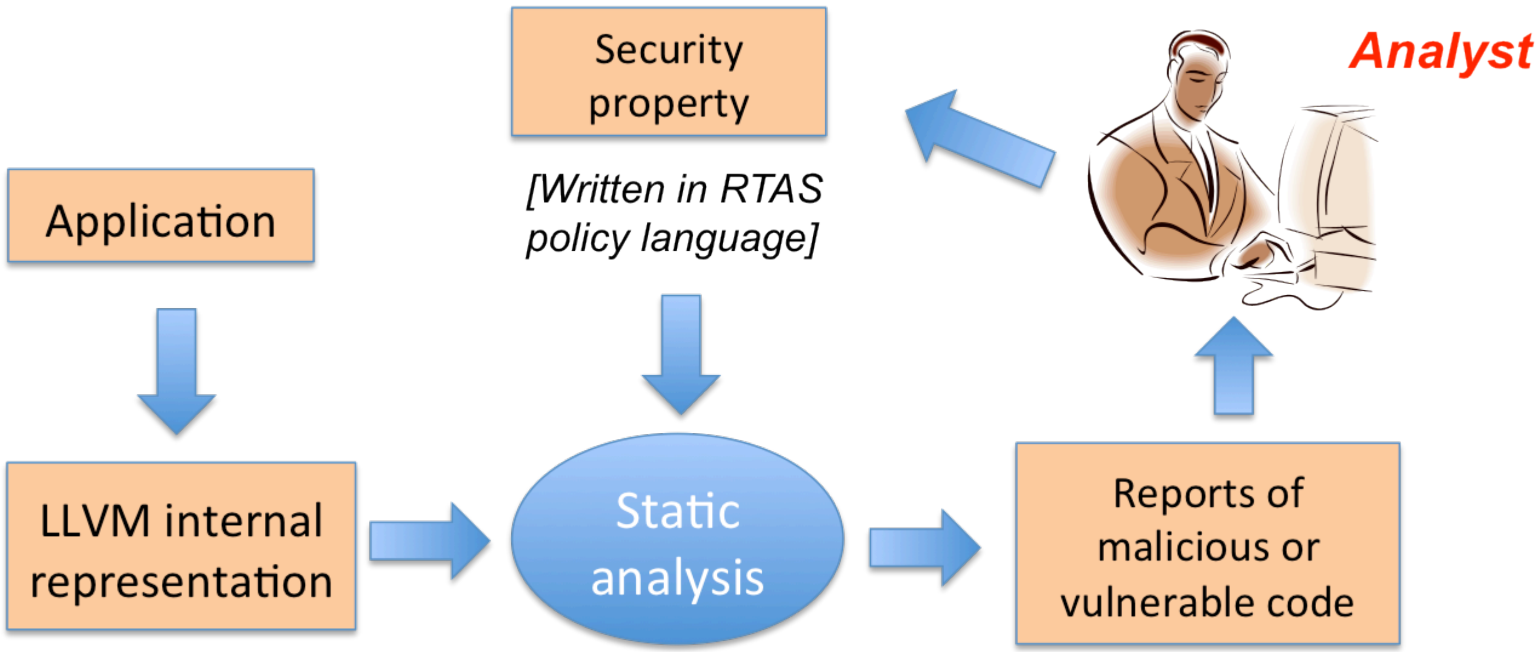


Figure source: Samsung Electronics, March 2013

Overall Approach



- Intra-application analysis
- Inter-application analysis

- Information flow analysis
- Control flow analysis

Taint Analysis: Assumptions

- All potential sources of tainted data and all sinks provided as input in security rules
- Static analysis of (source, sink) pairs
  - Can lead to false positives and false negatives
  - Dynamic analysis planned in Year 2 of project
- Output = prioritized list of vulnerabilities
- Can be used to model different security and privacy issues
  - Privacy leaks, unauthorized resource access, …

Input Rule Language

- Formulated in an XML file
- A rule identifies a (source, sink) taint pair
  - Identify source (API call or event callback)
  - Identify sink (API call)
  - Identify parameters of interest in source and sink

Source specification example

```
<source
    package="Tizen::Messaging"
    class="SmsManager"
    function="GetFullText"
    formals="int"
    return="Osp::Base::String"
    parameterId="0" />
```

parameterId = 0 indicates that return value is a taint source

Sink specification example

```
<sink
   function="write"
   formals="int,
         void *,
         size_t"
   return="int”
   parameterId=“2,3” />
```

parameterId = 2 and 3 are identified as taint sinks

Output of Taint Analysis

- XML file
- List of vulnerabilities
- For each vulnerability
  - Type of vulnerability
  - (source, sink) information
  - Ranking of vulnerabilities via distance metrics

Ranking of Vulnerability Reports

Approach:

- Rank vulnerabilities so that most likely errors appear closer to the top of the output list
  - A smaller rank indicates a higher priority
- Use a tunable cutoff threshold (e.g., top 100) on ranked output list, and return truncated list as official error report
  - Smaller threshold will decrease false positive rate but increase false negative rate

- Adaptation of approach proposed in “Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations”, Ted Kremenek, Dawson Engler, SAS 2003.

Ranking Heuristics

1. *Distance:* Evaluate source code distance between source and sink. Smaller distances should result in smaller ranks
2. *Number of conditionals:* The more conditionals in the information flow from source to sink, the more likely it is to be a false positive. A smaller number of conditionals should result in a smaller rank.
3. *Diagnosis effort:* The less time it takes a human to diagnose an error as a true vs. false positive, the smaller its rank should be. For example, intraprocedural errors are easier to diagnose than interprocedural errors, and should be given smaller ranks.

Taint Analysis

- Taint analysis lattice for values
  - Untainted = top
  - Tainted = bottom
  - meet(Tainted, Untainted) = Tainted
- Taint analysis will use similar approach to SSA-based sparse conditional constant propagation (SCCP) algorithm implemented in LLVM
- Important extensions
  - Use of control dependences to insert "pseudo uses"
  - Use of Array SSA form for efficient alias analysis

SSA-based Analysis Example

- SSA = Static Single Assignment
- Each definition is given a unique name
- SSA-based sparse analysis

```
x0 = ...
if (cond) {
    x1 = TaintSource();
    y1 = x1 + 1;
    TaintSink(y1);
} else
    TaintSink(x0);
x2 = phi(x0, x1);
```

Extensions to LLVM

- Pseudo-use insertion
  – build control dependence graph
  – Insert pseudo-use for each predicates
- Array SSA support
  – insert Array SSA def-use chains to connect all heap accesses
  – extend SSA analysis algorithms to Array SSA form

Example of inserting pseudo uses

x_0 = …

y_0 = …

if (x_0) {

    y_1 = 0; // pseudo_use(x_0);

}

else {

    y_2 = 1; // / pseudo_use(x_0);

}

y3 = $\Phi(y_1,y_2)$; // will include value flow from x_0

…

Array SSA Form and Related Work

	Control Flow	Array Subscripts	Renaming + Static Single Assignments
Classical Data Flow	General		
Scalar SSA	General		X
Array Dependence		Limited	
Array Data Flow	Limited	Limited	
Array SSA	General	General	X

Array SSA Form --- Definition ϕ

Definition Φ = *data merge* of array element modified in current def with array elements of previous def

Original Program

X[1:n] = . . .

. . .

X[k] = . . .

... = X[j]

Array SSA Form

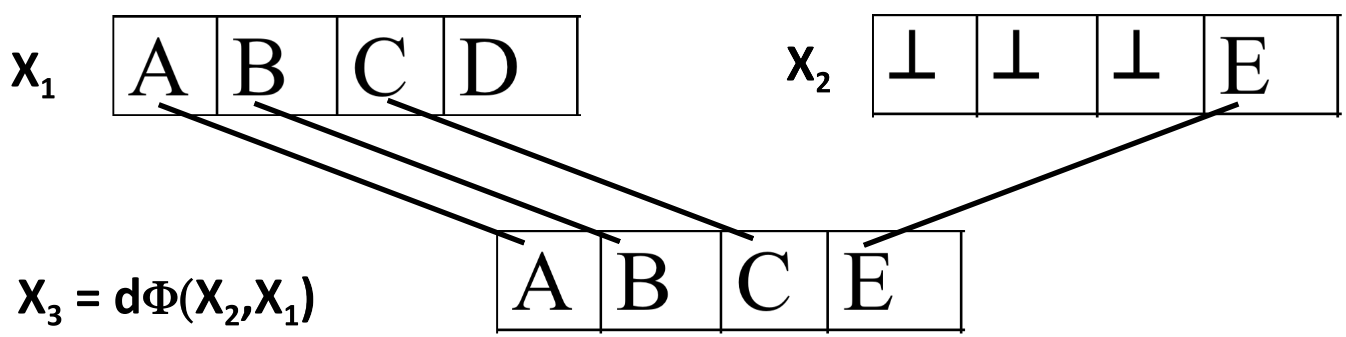
X_1[1:n] = . . .

. . .

X_2[k] = . . .

X_3 = dΦ(X_2,X_1)

... = X_3[j]



Conditional Constant Propagation using Array SSA form (Example)

i := 5	L(i) = 5 // lattice value
. . .	
if (i = 5) then	L(i=5) = TRUE
k := 3	L(k) = 3
X_1[k] := 99	L(X_1) = <(3,99)>
X_2 := dϕ(X_1,X_0)	L(X_2) = <(3,99)>
endif	
X_3 := mϕ(X_2,X_0)	L(X_3) = <(3,99)>
X_4[i] := 101	L(X_4) = <(5,101)>
X_5 := dϕ(X_4,X_3)	L(X_5) = <(3,99),(5,101)>
y := X_5[k]	L(X_5[k]) = 99

Extending Array SSA form to model objects and pointers

Introduce "Heap" array $\mathcal{x}$ for each field x

```
class Z { int x; };
...
Z a = new Z()
if (...) {
    a.x = 1
} else {
    a.x = 2
}
y = a.x
```

class Z { int x; };

...

a_9 = new Z( )

$x_1[a_9] = 0$

if (...) {

  $x_2\ [a_9] = 1$

  $x_3 = d\phi\ (x_1, x_2)$

} else {

  $x_4\ [a_9] = 2$

  $x_5 = d\phi\ (x_1, x_4)$

}

$x_6 = \phi\ (x_3, x_5)$

$y = x_6\ [a_9]$

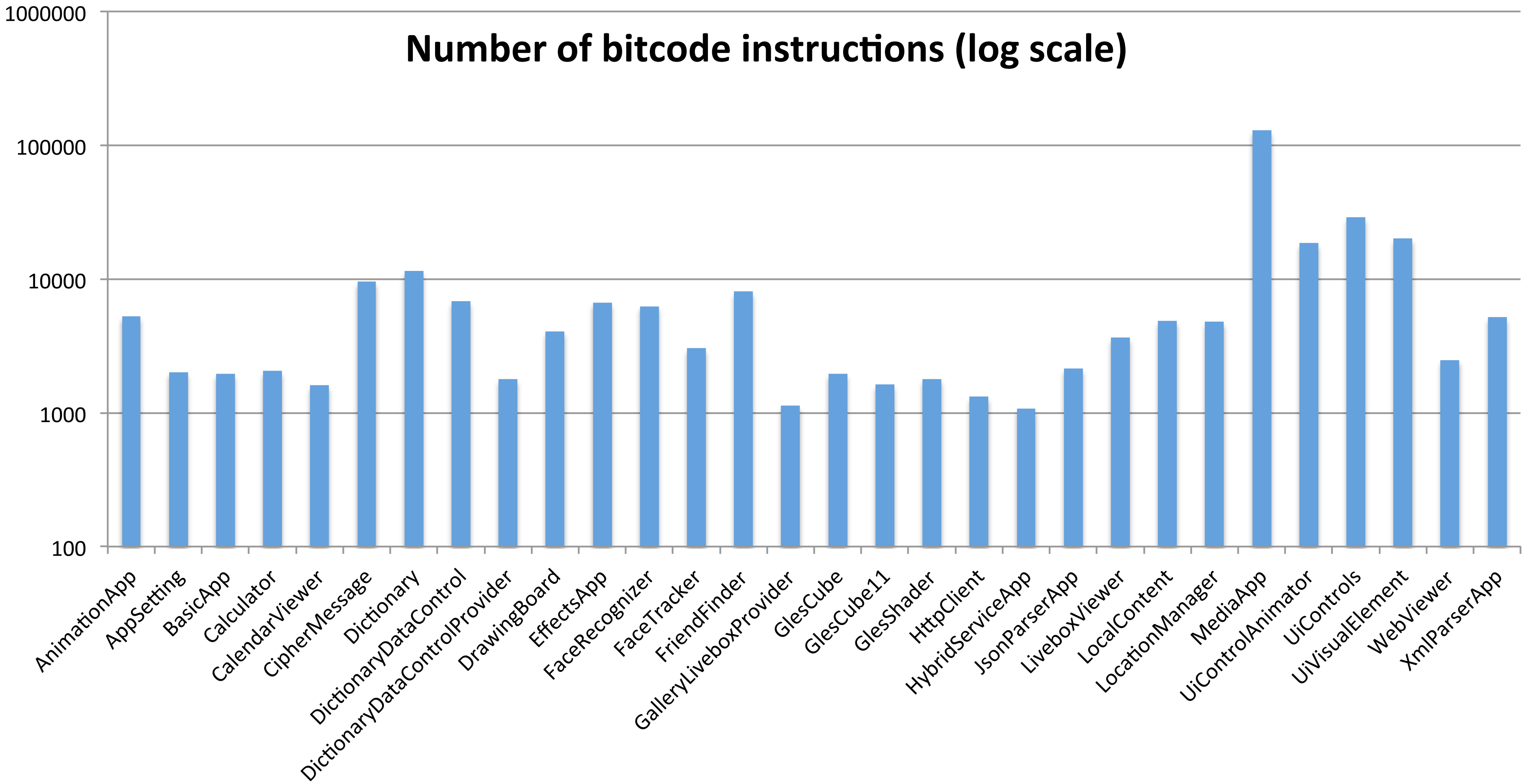
Results from using Taint Analysis for Privacy Leak Detection

- 30 Tizen applications
  - One true positive found
    - Privacy leak found in *FriendFinder* application
  - No false positives reported
  - No false negatives (to the best of our knowledge)
- Performance evaluation

Modeling Privacy Leak Analysis as Taint Analysis

- Security rules specify
  - Sources of privacy leaks (TaintSource)
  - Sinks of privacy leaks (TaintSink)
- Use SSA-based approach for efficiency of static analysis
  - Performance target is ~ 1000 bitcode instructions/second
- Rank privacy leaks with most likely reports closer to the top of the output
  - Cutoff threshold can tune false positive and false negative rates

Characterization of the 30 Tizen Applications



Example Privacy Leak Rule

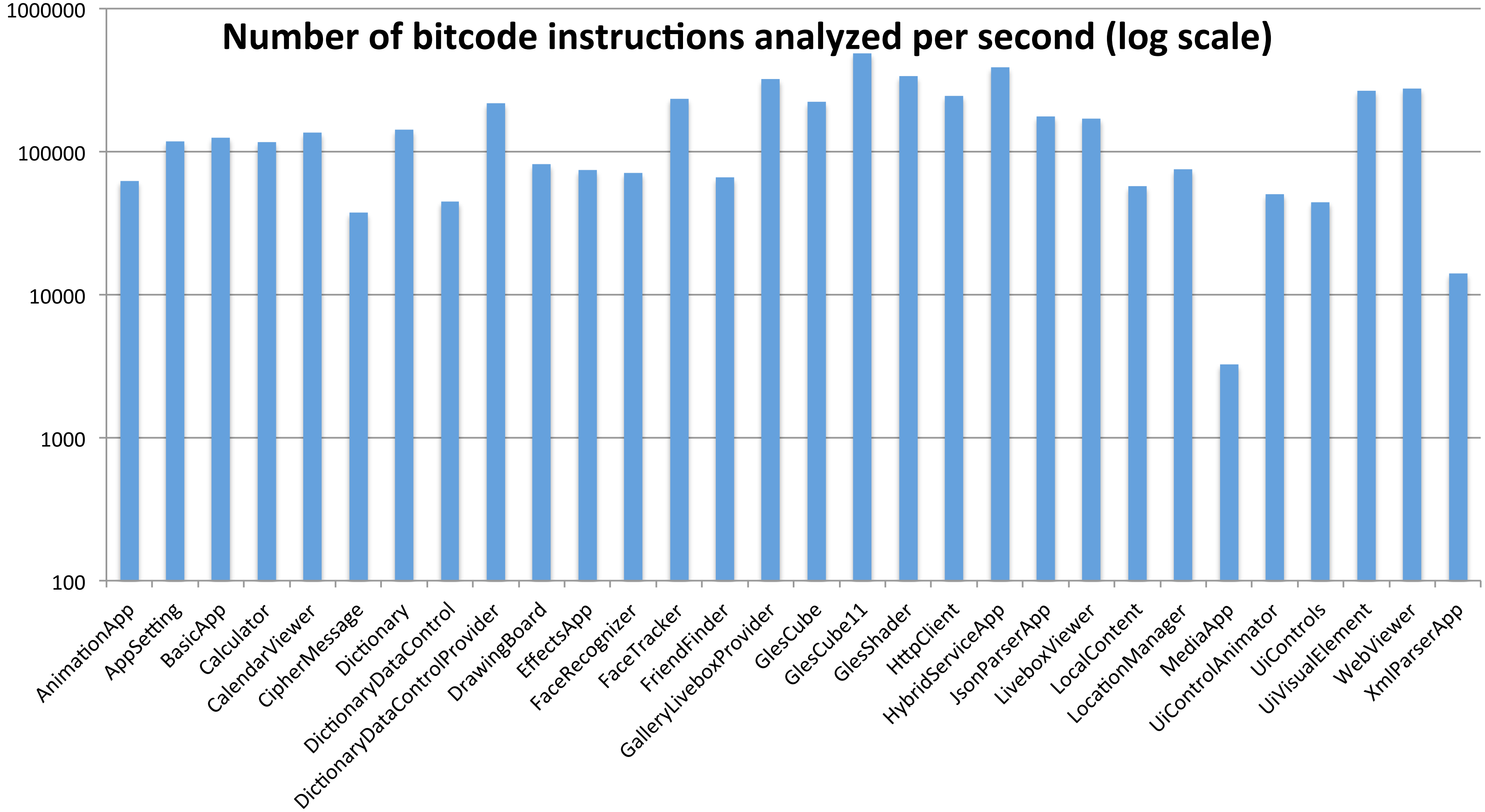
```
<source package=""
   class="ProfileManager"
   function="GetImagePathPtr"
   formals=""
   return="const wchar_t*"
   parameterId="0"/>

 <sink package="Tizen::Net::Bluetooth"
   class="BluetoothOppClient"
   function="PushFile"
   formals="Tizen::Net::Bluetooth::BluetoothDevice
const&amp;,
   Tizen::Base::String const&amp;"
   return="result"/>
```

Privacy Leak found in FriendFinder Example

```
...
W_char* str = GetImagePathPtr(); // taint source
...
String s = str;
BluetoothOppClient::PushFile(s); // taint sink
...
```


```
<vulnerability call_distance=”0" control_distance=”2">
  <source type="API" package="" class="ProfileManager"
      function="GetImagePathPtr" formals="" return="const wchar_t*”/>
  <sink package="Tizen::Net::Bluetooth" class="BluetoothOppClient"
      function="PushFile" formals="Tizen::Net::Bluetooth::BluetoothDevice const&amp;,
Tizen::Base::String const&amp;" return="result"/>
  <dataflow>
   <flow file="../src/ConnectionManager.cpp" line="142"/>
   <flow file="../src/ConnectionManager.cpp" line="144"/>
  </dataflow>
 </vulnerability>
```

Performance Evaluation



Conclusions & Future Work

- Summary
  - Security analysis of LLVM bitcode files can be performed efficiently and effectively
- Future Work
  - Refinement of ranking heuristics to reduce false positives and false negatives
  - Analysis of interprocedural events with callbacks
  - Dynamic analysis to complement static analysis
  - Approaches to automating generation of policy rules
  - Analysis of applications with web and native components