
COMPUTER SECURITY LAB
RICE UNIVERSITY

Security Analysis of LLVM Bitcode Files for Mobile Platforms

Dan S. Wallach (Rice University)

TIZEN™ DEVELOPER CONFERENCE 2013 SAN FRANCISCO

Preface

- This is ongoing research at Rice, supported by Samsung
- I don’t speak for Samsung

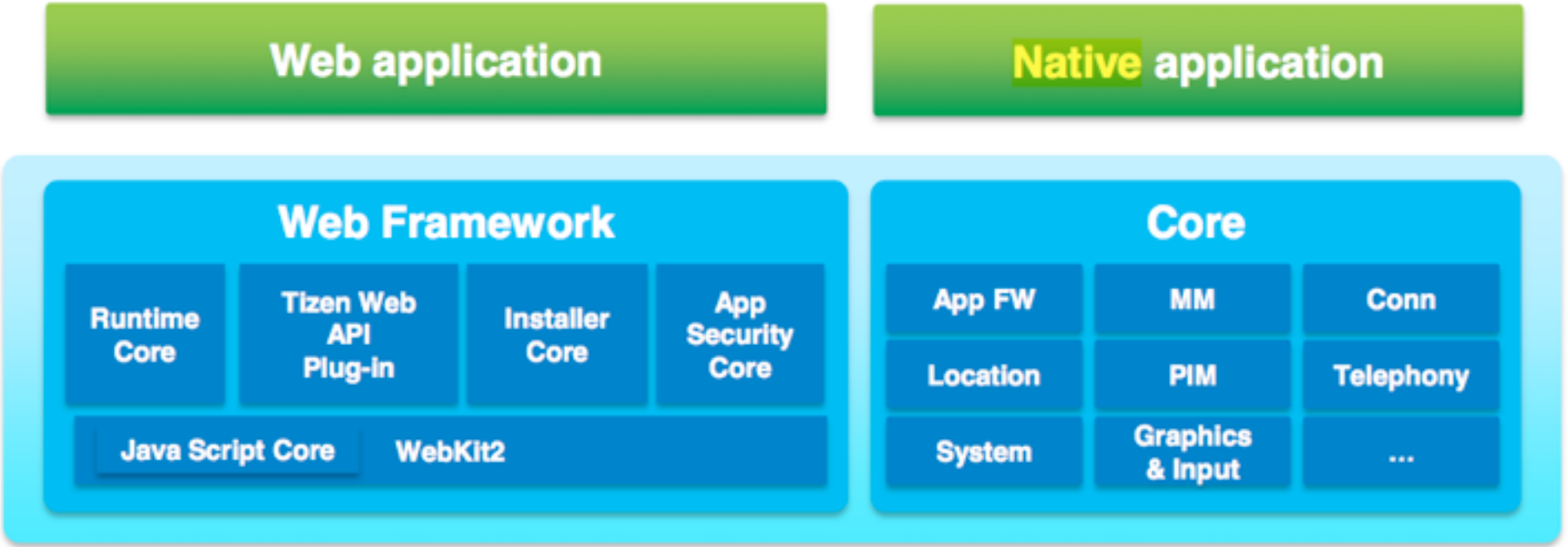
- **Core team at Rice**
  - Vivek Sarkar (PI)
  - Dan Wallach (Co-PI)
  - Michael Burke (Senior Research Scientist)
  - Jisheng Zhao (Research Scientist)
  - Deepak Majeti (PhD student)
  - Dragos Sbirlea (PhD student)
  - Bhargava Shastry (PhD student)
- **Additional contributors at Rice**
  - Keith Cooper
  - Swarat Chaudhuri

Security goals

- **Analyze apps submitted by developers, prior to deployment**
- **Static analysis (process applications without running them)**
  - Automation to assist a human analyst
- **Versus other models**
  - Apple iOS: unspecified process, apparently labor intensive and slow
  - Google Play Store: 100% automated, fast but problems get through

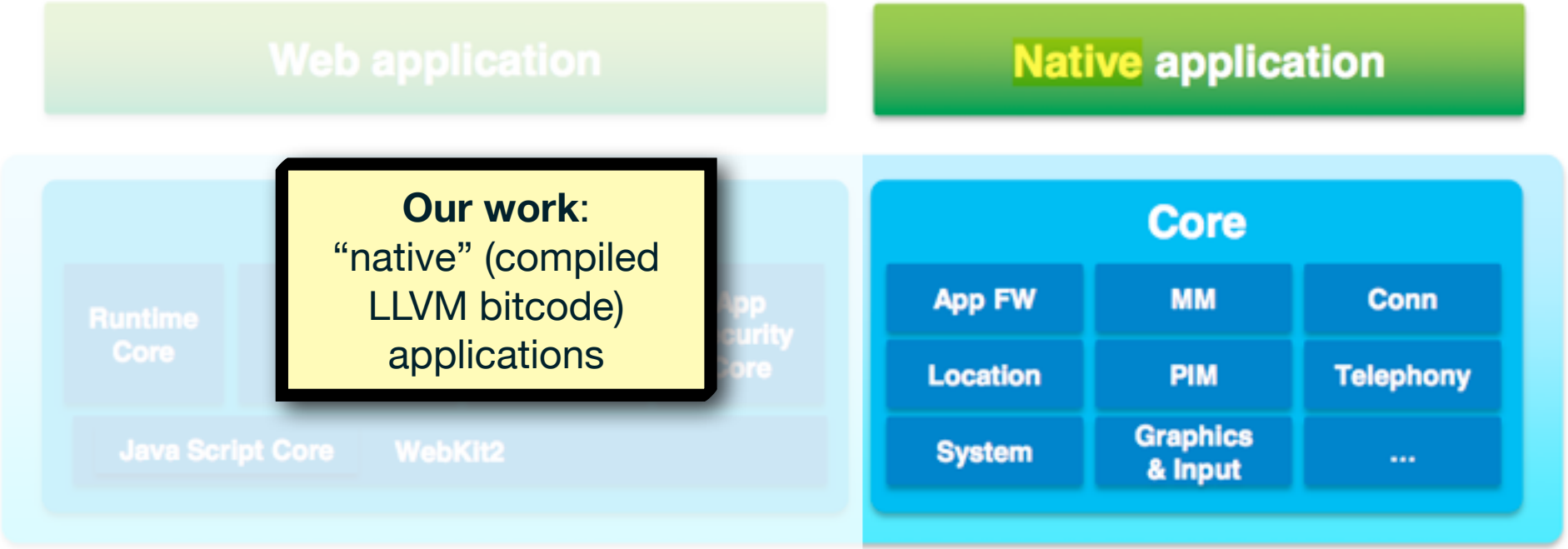
Web vs. native Tizen apps

Source: "Tizen Overview and Architecture", Seokjae Jeong, Samsung Electronics, Oct 2012

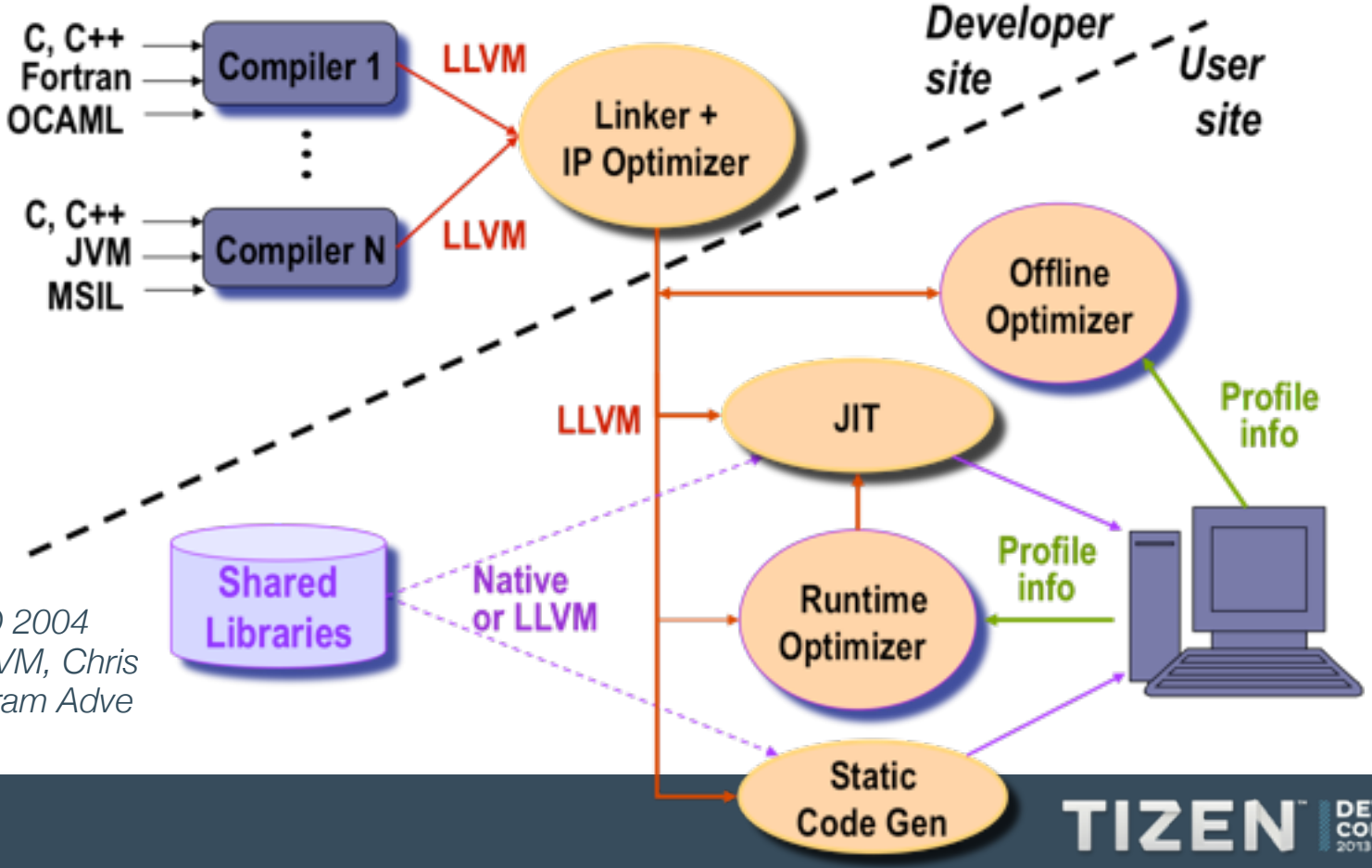


Web vs. native Tizen apps

Source: "Tizen Overview and Architecture", Seokjae Jeong, Samsung Electronics, Oct 2012



LLVM: A Low-Level Virtual Machine



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

LLVM is a great way to ship code

- **Designed to be analyzed / optimized**
  - Maintains high-level semantics of source code
  - Independent of any specific CPU architecture (unlike machine code)
  - Deals better with multiple source languages (unlike Java bytecode)
  - High runtime performance

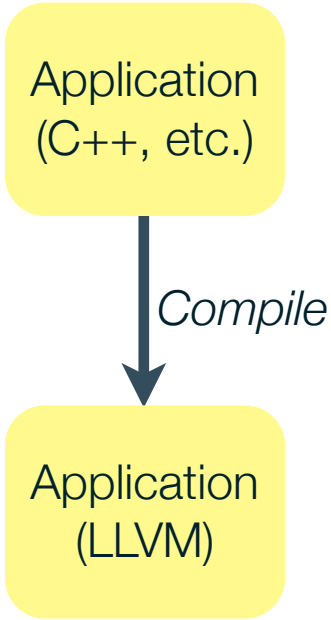
- **Open source / widely used in industry**
  - Notably adopted in Apple's Xcode toolchain

- **Straightforward to analyze for security purposes**

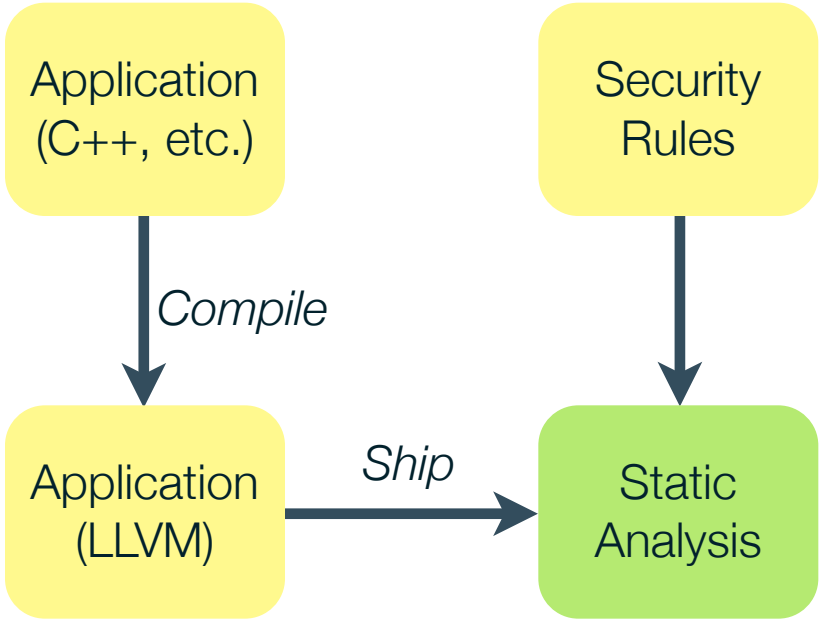
Overall Pipeline

Application
(C++, etc.)

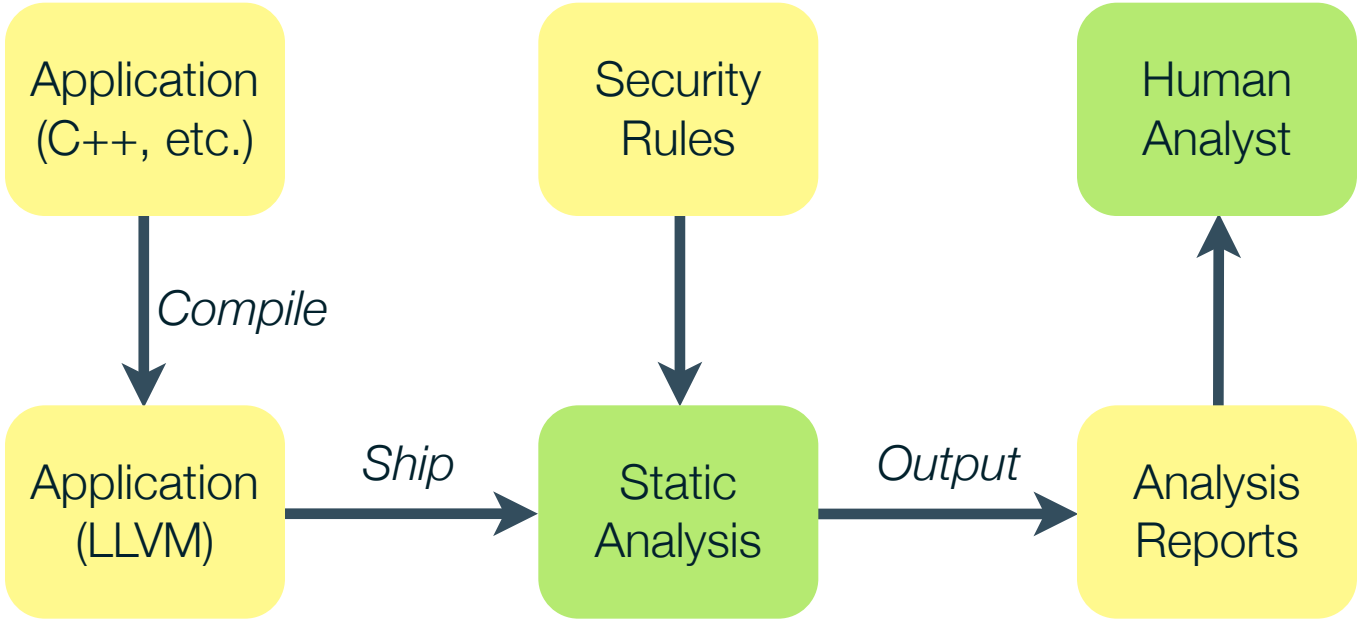
Overall Pipeline



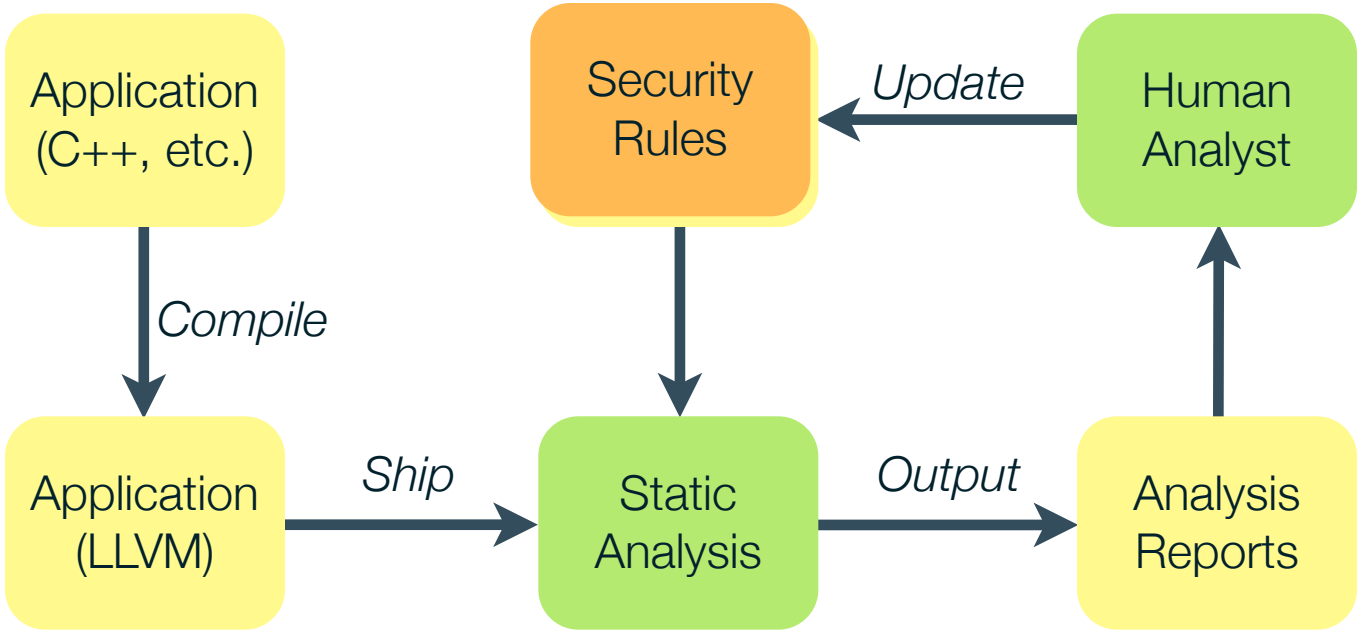
Overall Pipeline



Overall Pipeline



Overall Pipeline



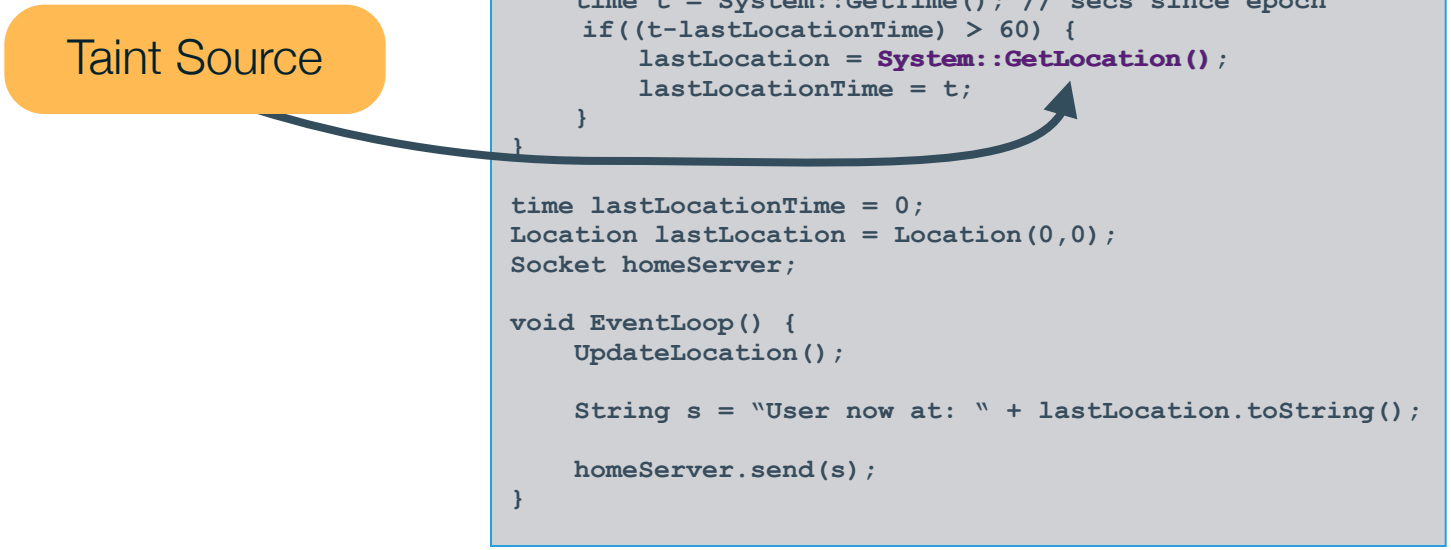
Possible analyses

- **Blacklist for APIs used**
  - Easy/fast to implement
- **Information flow**
  - Captures our intuition of policies we want to enforce
  - Example: “GPS to Internet”

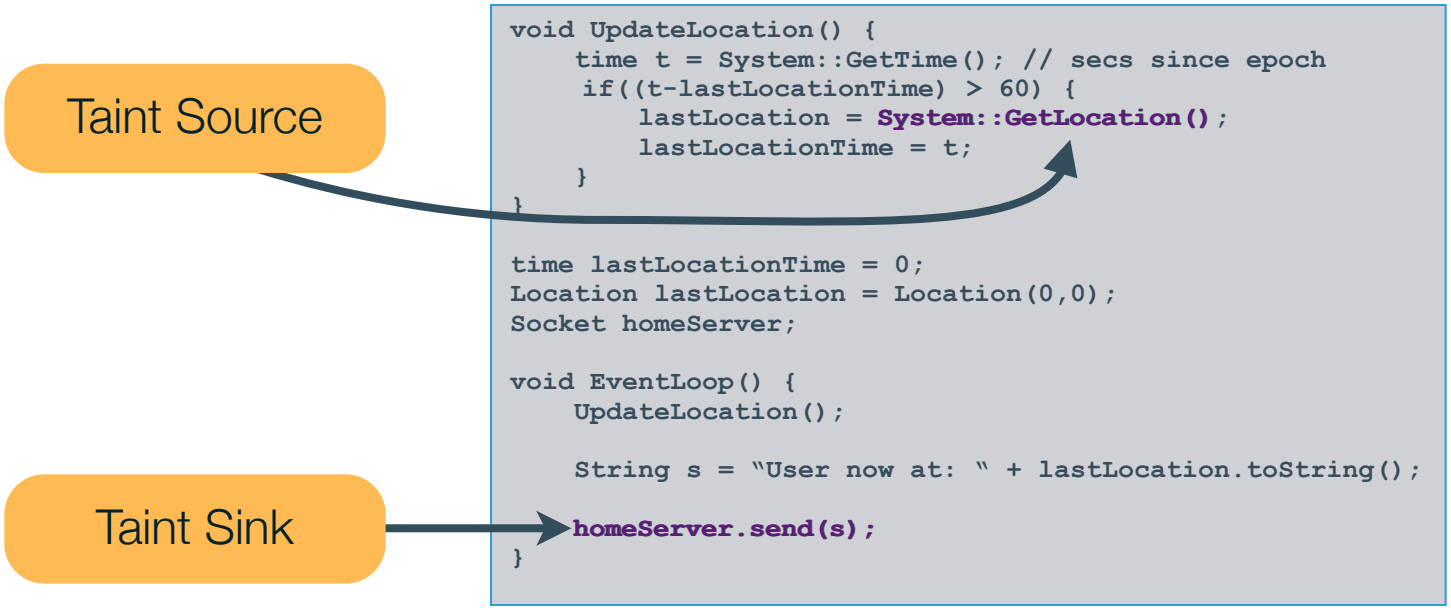
Example policy: “GPS to Internet”

```
void UpdateLocation() {
    time t = System::GetTime(); // secs since epoch
    if((t-lastLocationTime) > 60) {
        lastLocation = System::GetLocation();
        lastLocationTime = t;
    }
}

time lastLocationTime = 0;
Location lastLocation = Location(0,0);
Socket homeServer;

void EventLoop() {
    UpdateLocation();

    String s = "User now at: " + lastLocation.toString();

    homeServer.send(s);
}
```

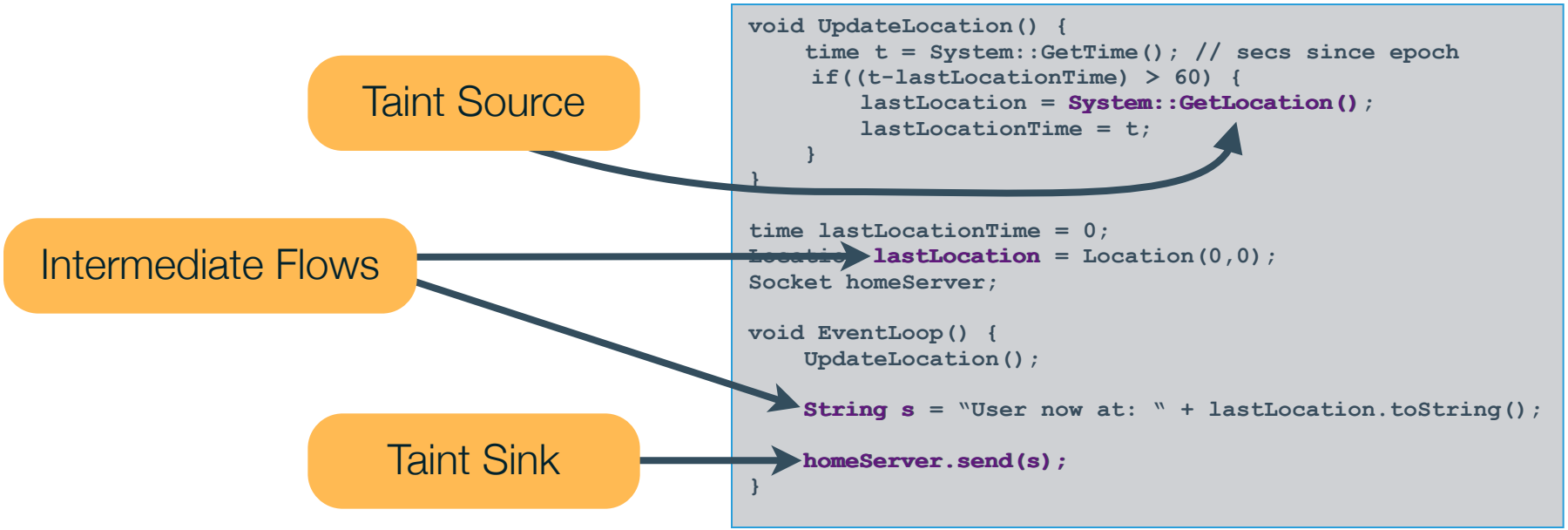
Example policy: “GPS to Internet”



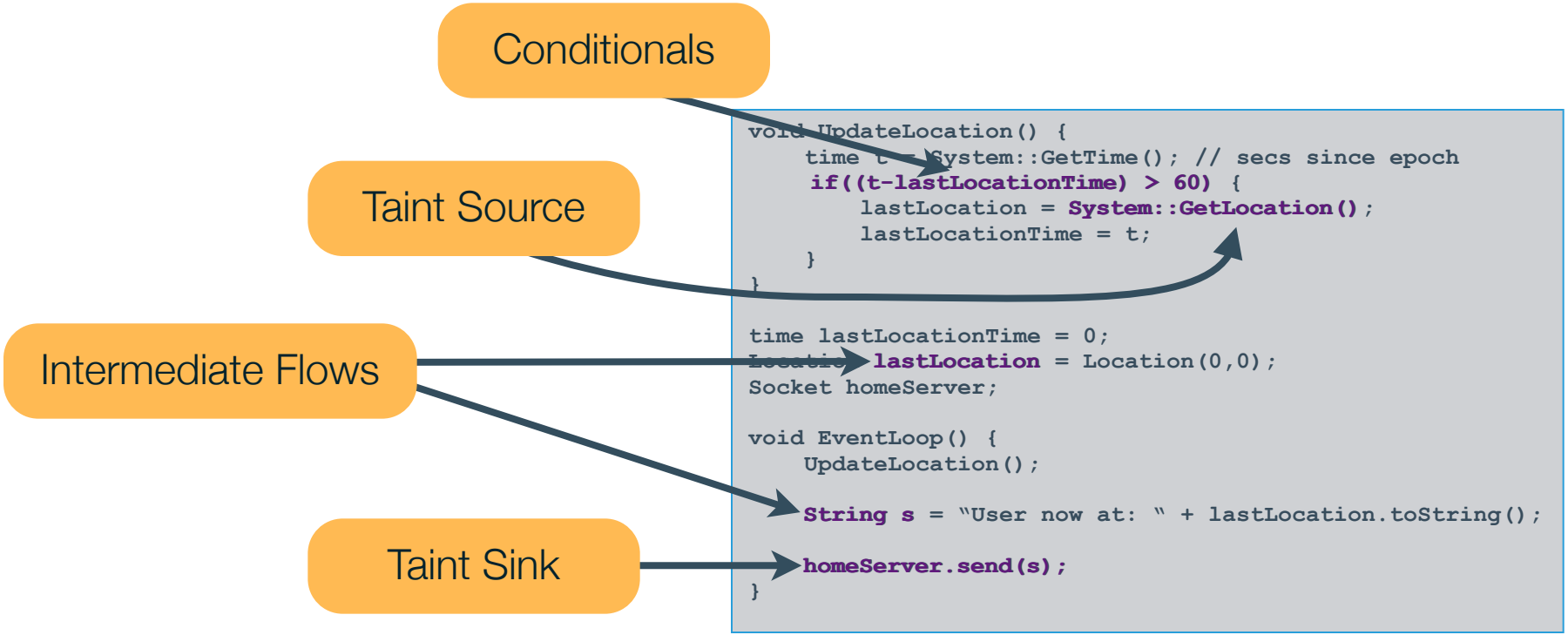
Example policy: “GPS to Internet”



Example policy: “GPS to Internet”



Example policy: “GPS to Internet”



Information flow complexities

- **Conditional flows**
  - Assume all branches taken
- **Pointer aliasing**
  - Don’t always know where a pointer points
- **OO method dispatch**
  - Don’t always know where a callsite will go
- **Challenge: Conservative analysis vs. false positives**

```
void UpdateLocation() {
    time t = System::GetTime(); // secs since epoch
    if((t-lastLocationTime) > 60) {
        lastLocation = System::GetLocation();
        lastLocationTime = t;
    }
}

time lastLocationTime = 0;
Location lastLocation = Location(0,0);
Socket homeServer;

void EventLoop() {
    UpdateLocation();

    String s = "User now at: " + lastLocation.toString();

    homeServer.send(s);
}
```

Information flow is a well-studied problem

- **Control-flow / data-flow analyses already used in compilers**
- **“Declassifying” operations to support special cases**
  - Example: system module trusted to give coarse location
- **False positives are an important challenge**
  - Conservatively flagging every possible flow would be overwhelming
  - Instead, pragmatic focus on “most likely” vulnerabilities

Prioritizing bug reports

- **Heuristics can help sort bug severity**
  - Shorter distance between source and sink ⇒ higher severity
  - More sensitive sources (GPS, contacts) ⇒ higher severity
  - More conditionals (harder to exploit) ⇒ lower severity
- **Heuristics, sources, and sinks are refined by analysts**
  - Learn from previous experience
  - Learn from (the inevitable) security exploits

SSA-based analysis example

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

```
x = …
if (cond) {
    x = TaintSource();
    y = x + 1;
    TaintSink(y);
} else
    TaintSink(x);
```

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 = ϕ(x0, x1);
```

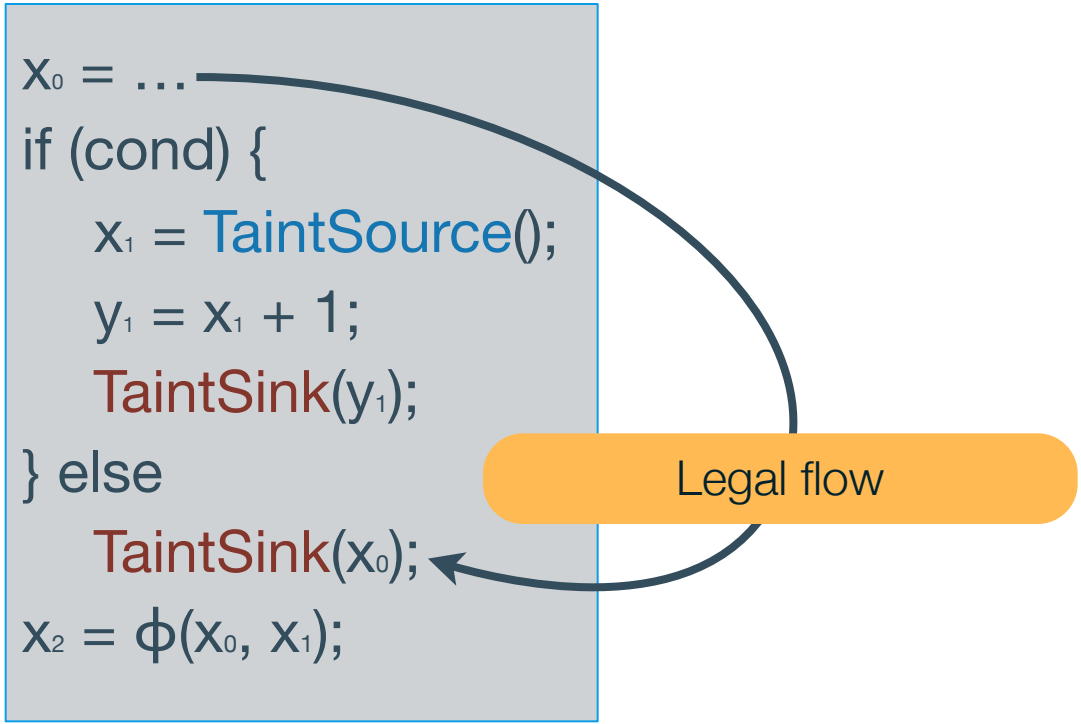
SSA-based analysis example

- **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 = ϕ(x0, x1);
```

SSA-based analysis example

- SSA = Static Single Assignment
- Each definition is given a unique name
- SSA-based sparse 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 = ϕ(x0, x1);
```

x_2 inherits taint from x_1

Pseudo-uses

- **Tainted conditional expressions**
  - Taint is applied to all subsequent values that depend on the condition
- **Similar issues**
  - Array storage
  - Method dispatch

```
x0 = TaintSource();
if (x0 == ...)
    y0 = ”yes”;
else
    y1 = ”no”;
y2 = ϕ(y0, y1);
TaintSink(y2);
```

Pseudo-uses

- **Tainted conditional expressions**
  - Taint is applied to all subsequent values that depend on the condition
- **Similar issues**
  - Array storage
  - Method dispatch

```
x0 = TaintSource();
if (x0 == ...)
    y0 = ”yes”;
else
    y1 = ”no”;
y2 = ϕ(y0, y1);
TaintSink(y2);
```

y_0 inherits taint from x_0

Pseudo-uses

- **Tainted conditional expressions**
  - Taint is applied to all subsequent values that depend on the condition
- **Similar issues**
  - Array storage
  - Method dispatch

x_0 = TaintSource();
if (x_0 == ...)
    y_0 = ”yes”;
else
    y_1 = ”no”;
$y_2 = \phi(y_0, y_1)$;
TaintSink(y_2);

y_0 inherits taint from x_0

y_1 inherits taint from x_0

Pseudo-uses

- **Tainted conditional expressions**
  - Taint is applied to all subsequent values that depend on the condition
- **Similar issues**
  - Array storage
  - Method dispatch

```
x0 = TaintSource();
if (x0 == ...)
    y0 = ”yes”;
else
    y1 = ”no”;
y2 = ϕ(y0, y1);
TaintSink(y2);
```

y_0 inherits taint from x_0

y_1 inherits taint from x_0

Forbidden flow

Under the hood: Taint analysis

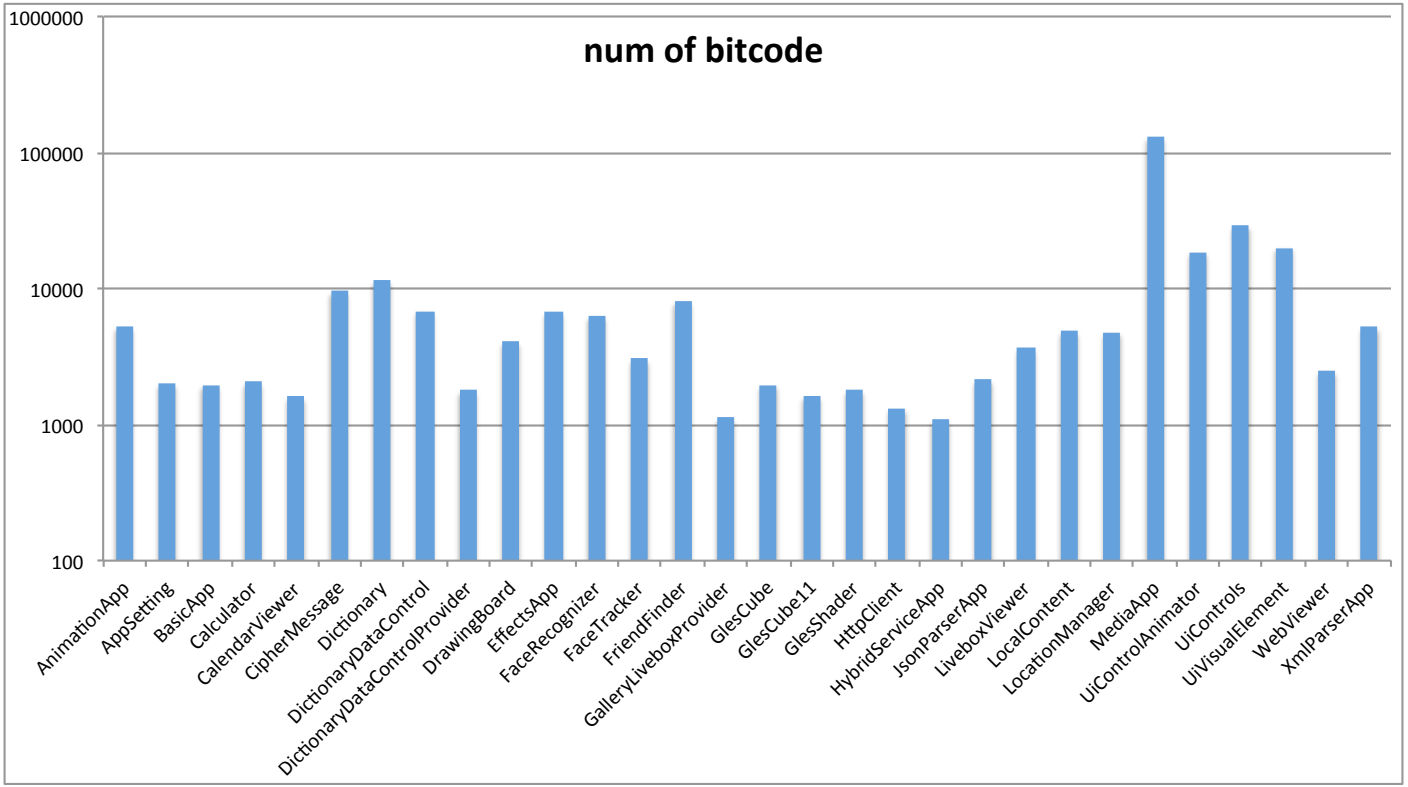
- **Simple lattice of taint values for every variable**
  - Untainted = top
  - Tainted = bottom
  - join(Tainted, Untainted) = Tainted
- **SSA-based sparse conditional constant propagation (SCCP) algorithm**
  - Already implemented in LLVM, extended by Rice for taint analysis
- **Important extensions**
  - Use of control dependences to insert “pseudo uses”
  - Use of Array SSA form for efficient alias analysis

Preliminary results

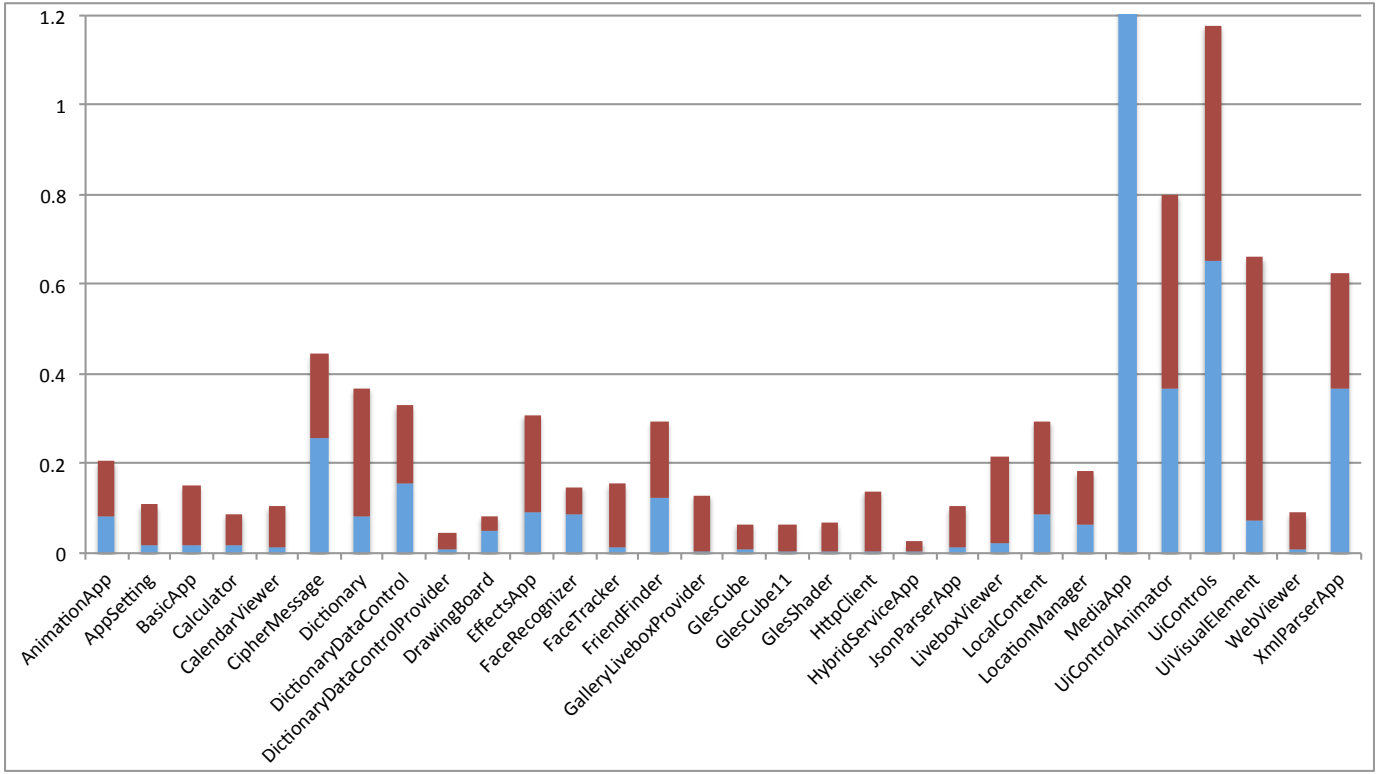
- **Analyzed 30 Tizen apps (from Samsung)**
- **Flagged one privacy leak (*FriendFinder*)**
  - Can transmit contact information over Bluetooth
- **No errors (false positives or false negatives)**

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

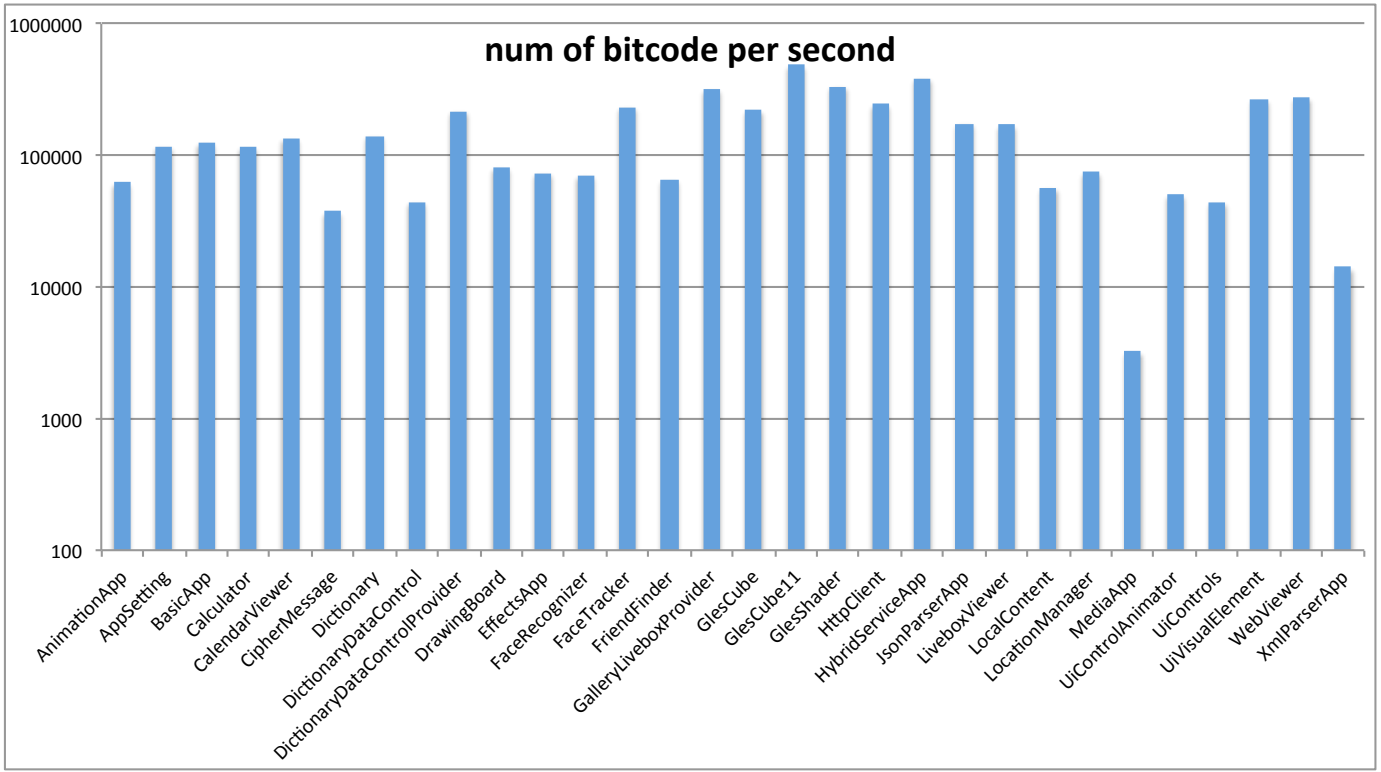
Preliminary performance



Preliminary performance



Preliminary performance



Future work

- **Full-system analysis (system libraries, kernel, etc.)**
  - Currently just analyzing the local code of the app
  - Special handling for callbacks

- **Dynamic analysis (virtual machine, runtime instrumentation)**

- **Tizen web apps**

TIZEN™

DEVELOPER CONFERENCE

2013

SAN FRANCISCO