Security Analysis of LLVM Bitcode Files for Mobile Platforms

Dan S. Wallach (Rice University)
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

Web vs. native Tizen apps


Our work: “native” (compiled LLVM bitcode) applications
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

Application (C++, etc.)

Compile

Application (LLVM)
Overall Pipeline

Application (C++, etc.) → Compile → Application (LLVM) → Ship → Security Rules → Static Analysis
Overall Pipeline

Application (C++, etc.) → Compile → Application (LLVM) → Ship → Static Analysis → Output → Security Rules → Output → Analysis Reports → Human Analyst
Overall Pipeline

- Application (C++, etc.)
  - Compile
- Application (LLVM)
  - Ship
- Security Rules
  - Update
- Human Analyst
  - Output
- Static Analysis
- Analysis Reports
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”

```cpp
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”

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

```c
void EventLoop() {
    UpdateLocation();

    String s = "User now at: " + lastLocation.toString();
    homeServer.send(s);
}
```

Taint Source
Example policy: “GPS to Internet”

```cpp
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”

```c
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);
}
```
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 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**

```cpp
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 $\Rightarrow$ higher severity
  • More sensitive sources (GPS, contacts) $\Rightarrow$ higher severity
  • More conditionals (harder to exploit) $\Rightarrow$ 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

```plaintext
x_0 = ... 
if (cond) {
    x_1 = TaintSource(); 
    y_1 = x_1 + 1; 
    TaintSink(y_1); 
} else 
    TaintSink(x_0); 

x_2 = \phi(x_0, x_1); 
```
SSA-based analysis example

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

\[ x_0 = \ldots \]
\[
\text{if (cond) } \{
\quad x_1 = \text{TaintSource}();
\quad y_1 = x_1 + 1;
\quad \text{TaintSink}(y_1);
\}\]
\[
\text{else}
\quad \text{TaintSink}(x_0);
\]
\[ x_2 = \phi(x_0, x_1); \]
SSA-based analysis example

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

```plaintext
X_0 = ...
if (cond) {
    x_1 = TaintSource();
    y_1 = x_1 + 1;
    TaintSink(y_1);
} else
    TaintSink(x_0);
X_2 = \phi(X_0, X_1);
```
SSA-based analysis example

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

\[
\begin{align*}
x_0 &= \ldots \\
\text{if} \ (\text{cond}) \ {\{ } \\
&\quad x_1 = \text{TaintSource}() \\
&\quad y_1 = x_1 + 1; \\
&\quad \text{TaintSink}(y_1); \\
\} \text{ else} \\
&\quad \text{TaintSink}(x_0); \\
x_2 &= \phi(x_0, x_1); \\
\end{align*}
\]

\(x_2\) 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

```
x_0 = TaintSource();
if (x_0 == ...)  
    y_0 = "yes";
else
    y_1 = "no";
y_2 = \phi(y_0, y_1);
TaintSink(y_2);
```
Pseudo-uses

• **Tainted conditional expressions**
  • Taint is applied to all subsequent values that depend on the condition

• **Similar issues**
  • Array storage
  • Method dispatch

\[
x_0 = \text{TaintSource}(); \\
\text{if} \ (x_0 == ...) \\
\quad y_0 = "yes"; \\
\text{else} \\
\quad y_1 = "no"; \\
y_2 = \phi(y_0, y_1); \\
\text{TaintSink}(y_2);
\]
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);

y₀ inherits taint from x₀

y₁ inherits taint from x₀
Pseudo-uses

- **Tainted conditional expressions**
  - Taint is applied to all subsequent values that depend on the condition

- **Similar issues**
  - Array storage
  - Method dispatch

```java
x0 = TaintSource();
if (x0 == ...)
    y0 = "yes";
else
    y1 = "no";
y2 = \phi(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 
...
RTAS identified one privacy leak in the FriendFinder application, and there are no false positives. In the ConnectionManager class, there is a function GetImagePathPtr that retrieves the path information as a string. In the same function, there is a BluetoothOppClient::PushFile function that takes the output string of GetImagePathPtr. This induces a privacy leak because the GetImagePathPtr API is obtaining a profile picture (i.e. file name) of the user and sending it to another device via the BluetoothOppClient::PushFile API.

Figure 1: The size of applications.
4.2.2 Performance

We ran RTAS on a quad-core Intel Xeon 2.66GHz workstation with 8GB of memory and running RedHat Linux (RHEL 5).

The measurements of execution time are shown in Figure 2. The Y-Axis is the execution time in seconds. We show both Total Execution Time (the total time for running RTAS on a given application), and the Analysis Time, which only includes the related LLVM analysis passes (pre-passes, interprocedural taint analysis). The bars in the figure show the total execution time. The blue parts gives the timing for analysis and red parts show Additional Execution Time, including LLVM initialization and file IO. Since the MediaApp is much larger than the others, it took 42.406 seconds for total execution, and the analysis took 39.693 seconds. For all of the applications, the Additional Execution Time is between 0.1 to 2 seconds, depending on the size of the input bitcode file.

![Figure 2: The execution time.](image)

Figure 3 shows the throughput (i.e. the number of bitcode instructions processed per second) for RTAS. Most of the applications have throughput around 10,000. The size of the application produced a negative impact on throughput, e.g. MediaApp's throughput is the slowest at 3263.8. There are 129,375 bitcode instructions in MediaApp, corresponding to 34,220 lines of source code in its .cpp and .h files, so the ratio of bitcode instructions to lines of source code for it is 3.78. RTAS processes 863.4 lines of C++ source code per second for MediaApp.
The RTAS output format is XML-based. Each vulnerability identified is specified as a `<vulnerability>` XML node. It contains sub-nodes for identifying the (source, sink) pair which information flow has identified in the analyzed application. The sources and sinks are API calls that have been specified as source or sink attributes of an input rule. The attributes `call distance` and `control distance` define a metric for the distance between the source and the sink in the application. See above for how these are used in the generation of output by the analyzer.
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**