**Dynamic Taint Tracking**

A technique to track the use of data items in a running system.

**Example: Tracking GPS data**

**A:** Define GPS to be a "taint source."

01: double lng = GetGPSLong();
02: double lat = GetGPSLat();

These variables become "tainted," because they receive values from a taint source.

**B:** Define "taint propagation" rules.

03: double newLng = normalize(lng);
04: double newLat = normalize(lat);

These variables become "tainted," because they are a result of other tainted variables.

**C:** Define "taint sinks."

05: sendToServer(newLng, newLat);

When "tainted" data is detected at a taint sink, we can allow, deny, or log the action that is taking place. Here, we know that GPS data is being sent to the server.

**Many Uses**

- Attest to the authenticity of sensor data. *YouProve (Gilbert '11)*
- Minimize exposure of sensitive data. *Clean OS (Tang '12)*
- Manage app data in logical units. *Pebbles (Spahn '14)*
- Track how apps use passwords. *SpanDex (Cox '14)*
- Improve energy efficiency of apps. *MobileHub (Shen '15)*

**Not just for detecting malware!**

**Key Limitation**

01: double lat = GetGPSLat();
02: boolean nearEquator = false;
03: if (lat >= 1 and lat <= 1) {
04:     nearEquator = true;
05: }

Should this be "tainted?"

In general, it is difficult to track taints through control flows.

... But not an insurmountable limitation ...

In some cases, it may not matter. *YouProve (Gilbert '11)*

In some, we can quantify the amount of information leaked through control flows. *SpanDex (Cox '14)*

And sometimes, we can even use ideas from static taint analysis to help us. *MobileHub (Shen '15)*

---

**Dynamic Native Code Taint Tracking**

Very expensive, 10x to 30x slowdown in previous work. *TaintCheck (Newsome '05), Dytan (Clause '07)*

Slowdown is due to full-program emulation.

- Each program instruction is emulated.
- Taint propagation is done after emulation of each instruction.

Performance can be improved via **Selective Taint Tracking**.

**Demand Emulation (Ho '06)**

A) Use page protections to trap access to tainted data in memory.
B) Let the program run normally.
C) When it traps because of a tainted data access, emulate it.
D) Let it run normally when it no longer handles tainted data.

**Insight: Amortize overhead over app's lifetime.**

---

**Our Work**

**Goal:** Perform dynamic native code taint tracking on real devices.

**Problem**

- Performance is critical. (So we want selective taint tracking.)
- Mobile apps are inherently multi-threaded. (Page protections apply equally to all threads.)
- What if only one thread needs to access tainted data? (Need to emulate every thread!)

**We Need Thread-local Emulation!**

**Thread-local Emulation**

- Only emulate threads handling tainted data.
- All other threads run at non-emulated speeds.
- Particularly important in mobile apps. (as the UI thread has to run quickly.)

**Key Challenge**

How do we discriminate memory accesses on a per-thread basis? (Since only emulated threads may freely access tainted data.)

**Current Prototype**

- Integrates with TaintDroid. (Tracks taints through 3rd-party native libraries.)
- Handles a subset of ARM instructions.
- Taint tracking on new Android is done via app rewriting.

**Ongoing Work and Questions**

- What is the performance benefit of thread-local emulation?
- Will we affect correctness by slowing down some threads?