Software Protection: Static Security Analysis

Shuai Wang

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Some slides on taint analysis are from Edgar Barbosa.

Automatic Vulnerability Detection

Find a needle in a haystack

Static Automatic Vulnerability Detection Techniques

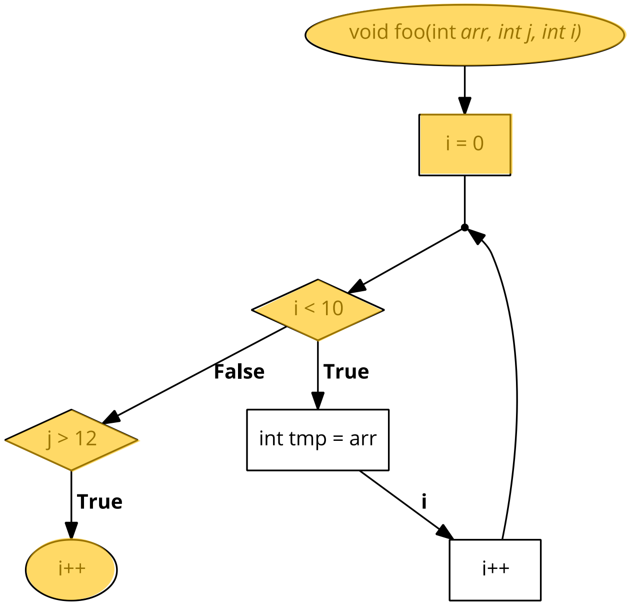
- Dynamic vs. static methods ← this time.
- Taint analysis (information flow) ← this time
- Concolic execution
- Symbolic execution
- Type system
- Formal verification
- Sound vs. complete ← The fundamental property of almost all static security analyzer (this time).

Dynamic security methods

- Fuzzing: Stress the software with random/crafted input before deployment, hopefully to trigger bugs and fix them
  - Talked last week.
  - Mostly mature.
- Sanitization: monitoring the execution of the program and capturing abnormal behavior that indicates that an attack is about to happen (or happening).
  - Still some open challenges (e.g., too slow).
  - Not covered in this course…

Dynamic Security Fuzzing

Given a control flow graph of the program.



(a) Dynamic methods can only assert an executed path a time, therefore can have *false negatives*.

```
if (a) {
  // do something
}
if (b) {
  // do something
}
...
if (z) {
  // do something
}
```

How many different paths in this program?

$2^{26} = 67108864$

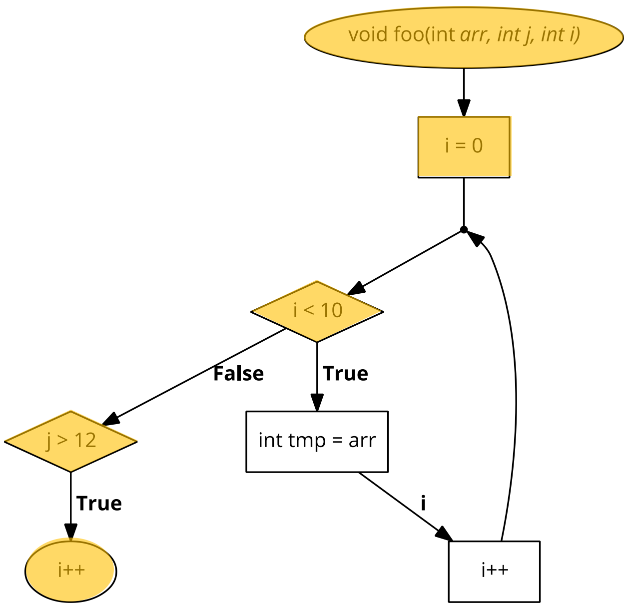
In principle, it's not guaranteed to fully cover the entire program.

Static Analysis, in a Security Context

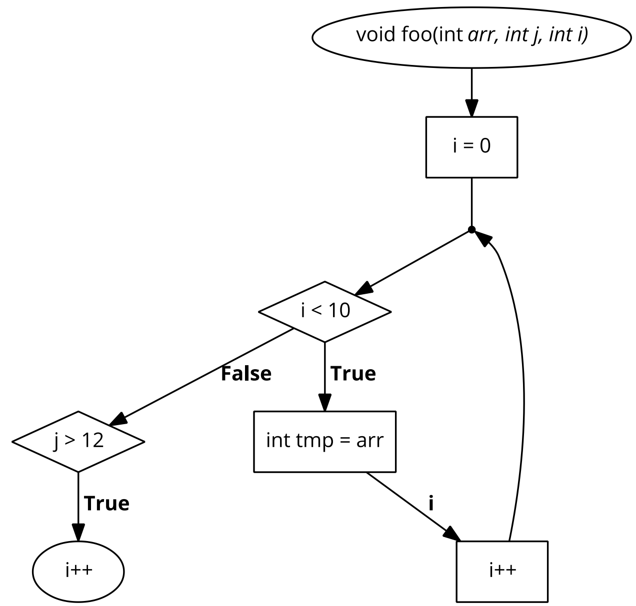
- Looking for defects in the source code without running it
  - Taint analysis
  - Formal verification
  - Symbolic execution
- Different methods but essentially with the same goal
  - Statically modeling program behavior in terms different aspects
    - Taint → critical information flow propagation
    - Type → mostly syntax-level data annotation
    - Symbolic execution → semantics-level info regarding symbolic values

Static vs. Dynamic

Given a control flow graph of the program.



(a) Dynamic methods can only assert an executed path a time, therefore can have *false negatives*.



(b) Static methods can analyze the whole program, but could have false positives. ← *explain a typical FP case in a moment.*

Static methods analyze the entire graph typically using the working list framework, whose algorithmic complexity is O(n).

Terminology Particularly Important for Static Analysis

A security analysis tool finds a vulnerability in the program.

True Positive: the found "vulnerability" is indeed a true vulnerability
False Positive: the found "vulnerability" is NOT a vulnerability
False Negative: a true vulnerability is missed.
True negative: that's fine...

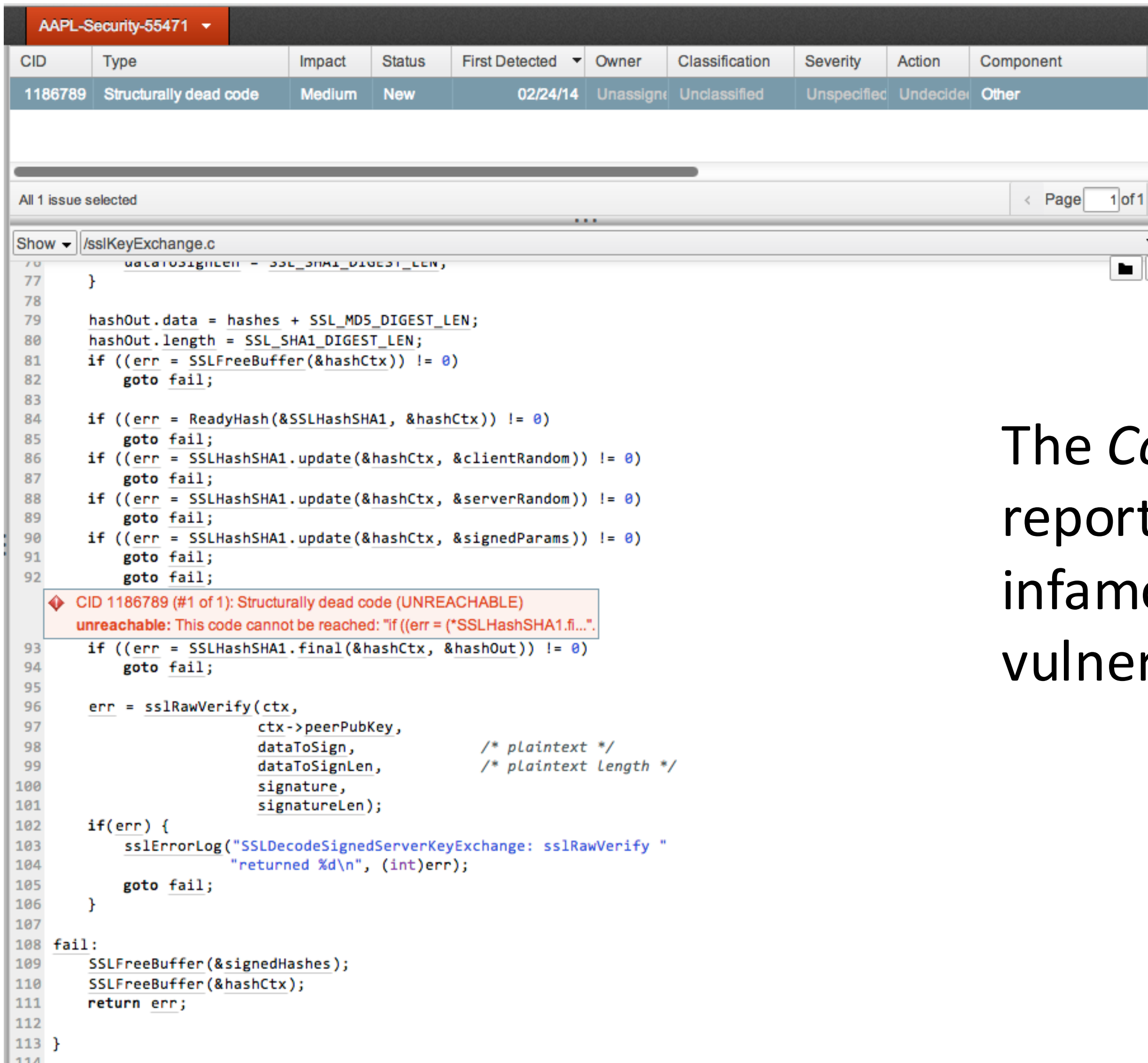
Static vs. Dynamic

Static analysis *can be* comprehensive enough to prove the absence of bugs/vulnerabilities.

> *"Prove, with a machine-checked proof using a deductive system, that a program implementation satisfies a logical specification."*
>
> — Andrew Appel

This is the ideal case (not easy to achieve; usually introduces many false alarms), will explain more in the soundness vs. completeness module.

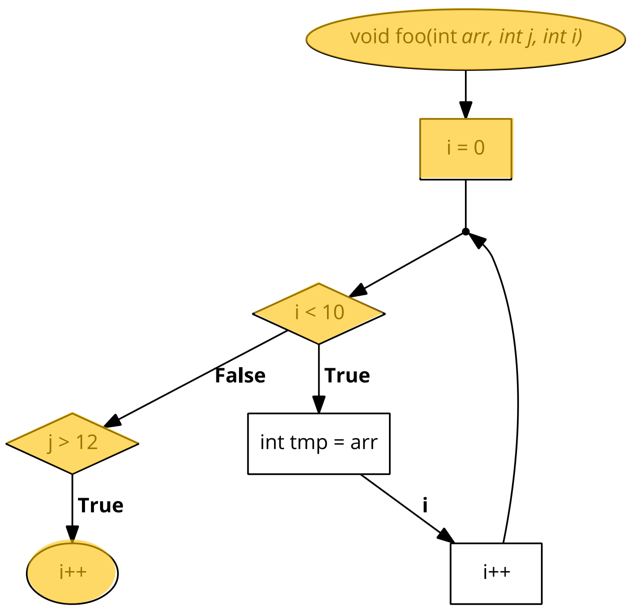
A Working Static Analyzer



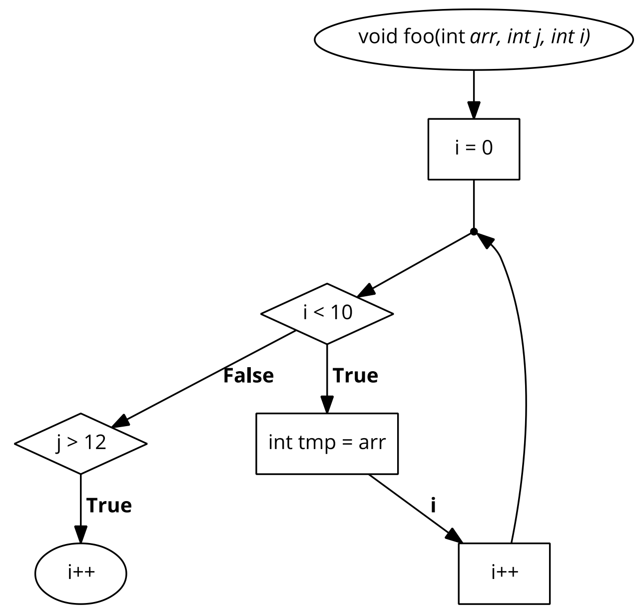
The *Coverity* static analyzer reports a code defect (the infamous Apple "goto fail" vulnerability)

Static vs. Dynamic

Given a control flow graph of the program.



(a) Dynamic methods can only assert an executed path a time, therefore can have *false negatives*.



(b) Static methods can analyze the whole program, but could have false positives. ← *explain a typical FP case now.*

Static methods analyze the entire graph typically using the ***working list framework****, whose algorithmic complexity is O(n).*

False Positives – Limits of Static Analysis

```
int x = 0;
int y = 2;

void foo()
{
   char s[10];
   if (x + y ==2) {
      s[20] = 0;
   }
}
```

```
d:\StaticAnalysis>cppcheck example.cpp
Checking example.cpp...
[example.cpp:7]: (error) Array 'a[10]' accessed at index 20, which
                 is out of bounds.
```

Access out of bound??

False Positives – Limits of Static Analysis

```
int x = 0;
int y = 3;

void foo()
{
    char s[10];
    if (x + y ==2) {
        s[20] = 0;
    }
}
```

```
d:\StaticAnalysis>cppcheck example.cpp
Checking example.cpp...
[example.cpp:7]: (error) Array 'a[10]' accessed at index 20, which
                 is out of bounds.
```

Access out of bound??

- For x + y != 2 ➔ false positive!
- But analyzer cannot be sure about x and y values!!! ← for the scalability matters.
  - However, it's **not** a bad idea to be **conservative** (sound vs. complete)

Static and Dynamic Security Analysis

- Dynamic methods ← must run the software
  - Fuzz testing ← find security flaws with testing
  - Sanitiziertion <-- find security flaws by monitoring daily usage
- Static analysis ← do not need to run it
  - **Taint analysis ← today**
  - Symbolic execution
  - Formal verification
  - Type systems
  - …

Taint Analysis in the security context

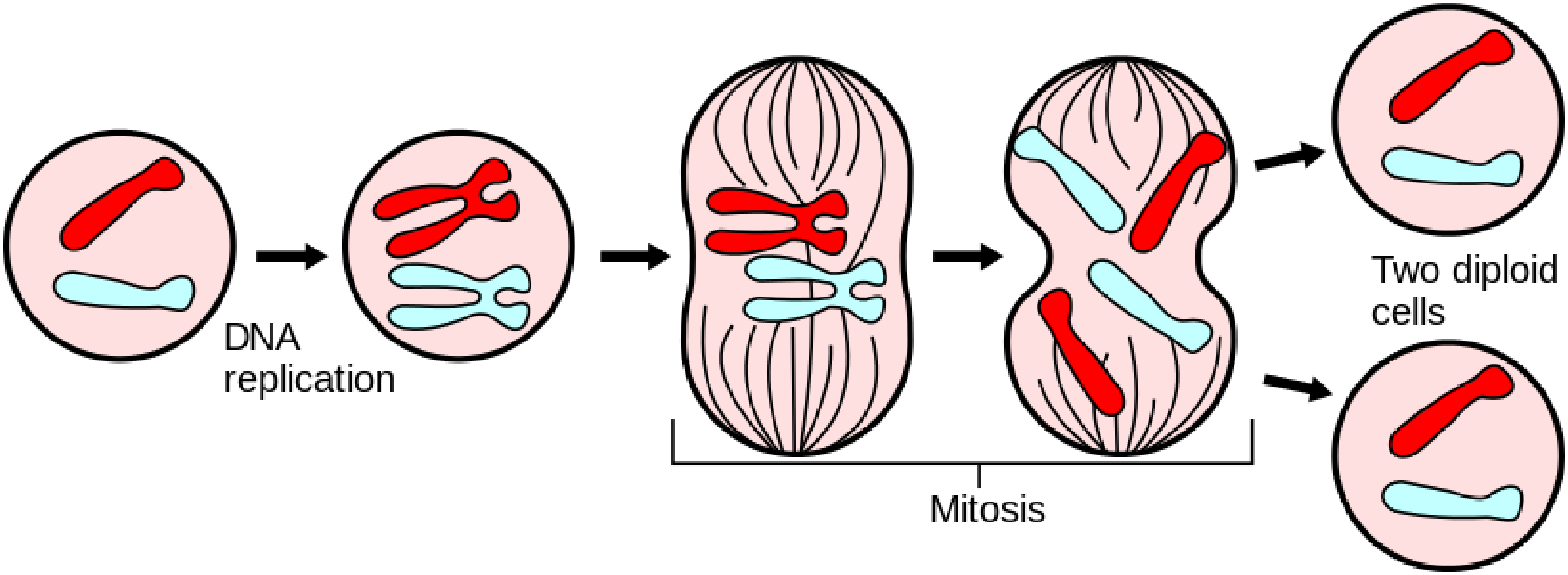
- Is it possible to measure the level of (unexpected) influence that external data have over some software?
  - Buffer overflow attack; format string attack; …
- And also measure (unexpected) influence from sensitive data of a software on the external environments
  - ← can be observed by attackers
  - Password leakage, private data leakage

Information Flow Tracking

Information Flow

- Data is being copied and modified all the time in a program. In another words, *"information" is always moving*.
- Note that information flow is more general than just "data flow".
  - A piece of data being used by some statements.
  - A piece of data affects the execution of certain if/else branch.
  - A piece of data affects the interaction with some system calls.
  - A piece of data affects the usage of some network/hardware resources
  - …
- Track Information Flow Analysis with Software Taint Analysis

What is "taint analysis"?



Analogy to mitosis in biology.

Mitosis*: "a type of cell division that results in two daughter cells."*

Information Flow Analysis with Software Taint Analysis

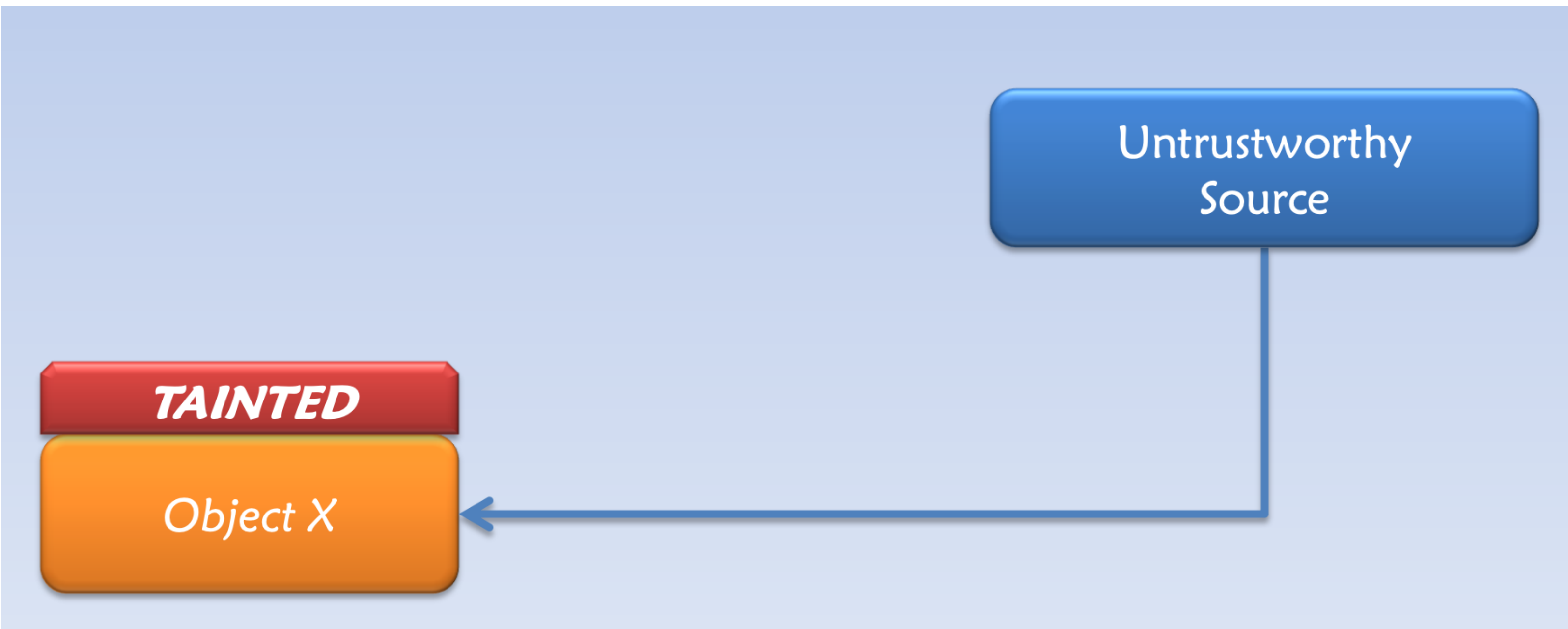
- Track Information Flow Analysis with Software Taint Analysis.
  - Taint analysis is very powerful to find software flaws, attack vectors, privacy disclosure, etc.
  - More importantly, it is very easy for implementation
  - And also very efficient
    - CodeQL

Software Taint Analysis – Three Step Approach

- From a holistic view, three components to define a software taint analysis
  - Taint source
  - Taint propagation
  - Taint sink

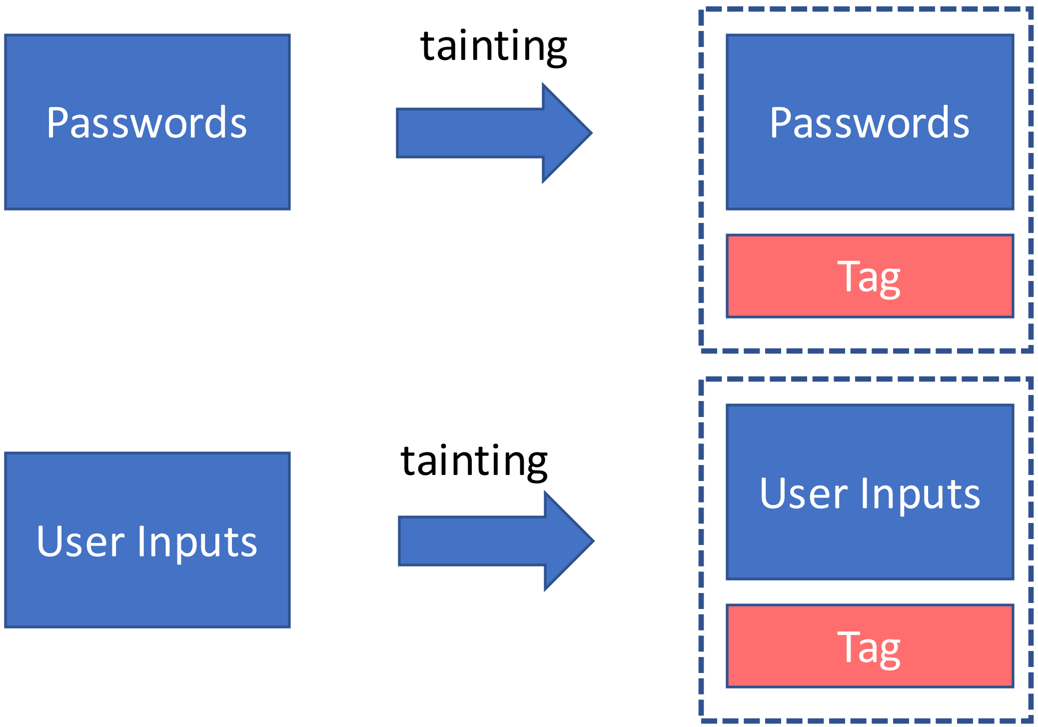
Software Taint Analysis – Taint Source

- If the source of the value of the object X is untrustworthy or sensitive, we start by tainting X.



Software Taint Analysis – Taint Source

- To "taint" user data is to insert some kind of tag or label for each object of the user data.
  - The tag allow us to track the influence of the tainted object along the execution of the program.



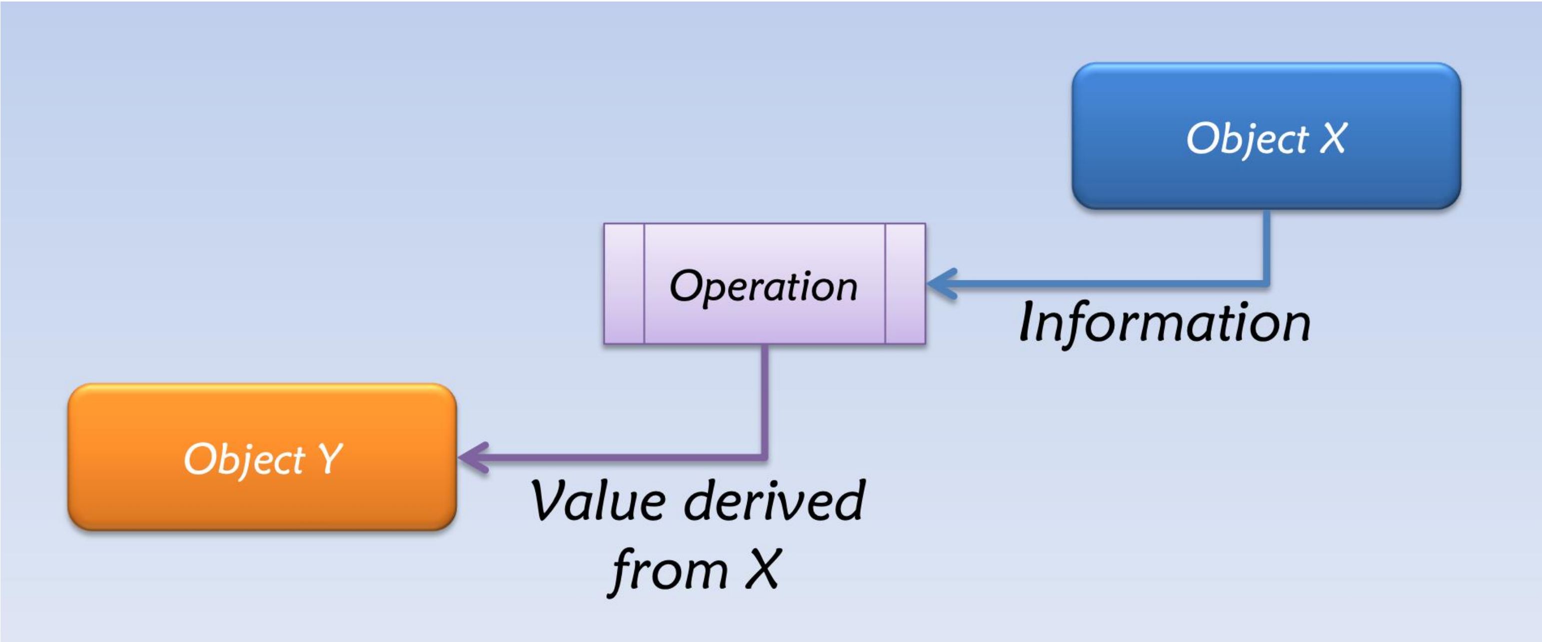
"tag" indicates this data is critical in the context of Cybersecurity.

Software Taint Analysis – Taint Source

- Generally any untrusted or sensitive information
  - Files (*.mp3, *.pdf, *.svg, *.html, *.js, …)
  - Network packages (HTTP, UDP, DNS, ...)
  - Keyboard, mouse and touchscreen input messages
  - Webcam
  - USB

Software Taint Analysis – Taint Propagation

- “Information flows from object x to object y, denoted x → y , whenever information stored in x is transferred to, object y”



Software Taint Analysis – Taint Propagation

- Usually, we need to define at least one taint propagation policy for each computation statement in the software.

Assignment	m = e	→	m = e	
Arithmetic & Logic	m = $e_1 \diamond e_2$	→	m = $e_1 \diamond e_2$	
Arithmetic & Logic	m = $e_1 \diamond e_2$	→	m = $e_1 \diamond e_2$	
Arithmetic & Logic	m = $e_1 \diamond e_2$	→	m = $e_1 \diamond e_2$	
Memory Load	m = load(addr)	→	m = load(addr)	→ Implicit information flow
Memory Store	store(addr,e)	→	memory content is tainted	
Memory Store	store(addr,e)	→	memory content is tainted	→ Implicit information flow
Conditional	if (e) $\{b_{then}\}$ else $\{b_{else}\}$)	→	if (e) $\{b_{then}\}$ else $\{b_{else}\}$	→ Implicit information flow
Loop	while (e) {b}	→	while (e) {b}	→ Implicit information flow
Function Call	ret = fun(e)	→	fun will be executed with tainted inputs	
Function Return	return e	→	ret = fun(...)	

Sample taint propagation policies.

- Taint operator is transitive
  - X → Y and Y → Z ➔ X → Z

Software Taint Analysis – Taint Sink

- Would some sensitive/untrusted program points being affected by critical information?
  - A buffer overflow bug
  - A network API
  - A hardware device ← later in "side channel" module
  - …..
- Sensitive program points can be defined as "taint sink".

```
int send_msg_out(char* data)
```

If "data" is tainted, what does that imply?

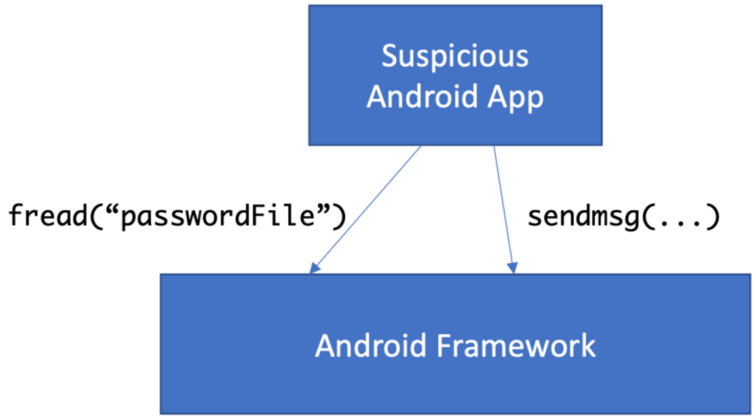
Software Taint Analysis – Application

- Exploit detection
  - If we can track user data, we can detect if nontrusted data reaches a privileged location
  - – format string, buffer overflows, …
- Perl tainted mode (a "dynamic" taint analysis module)
  - Before execution of any statement, the taint analysis module checks if the statement is tainted or not! If tainted issue an attack alert!

Software Taint Analysis – Malicious Behavior Analysis

Coarse-grained Behavior: Malware in Android

- Log the system call sequences as a way to reflect “malicious” behaviors.
  - How?

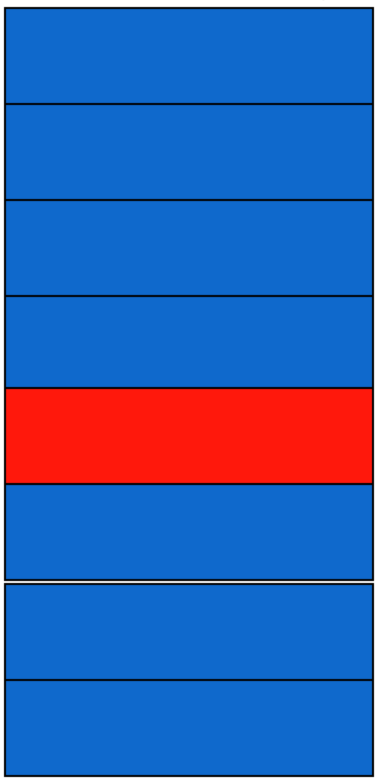


Sort of Malicious, although in practice we need to do more.

Yes, we need taint analysis!

Software Taint Analysis – Boost Fuzz Testing

- Critical questions in designing an efficient fuzzer:
  - Which input byte affect a path condition?



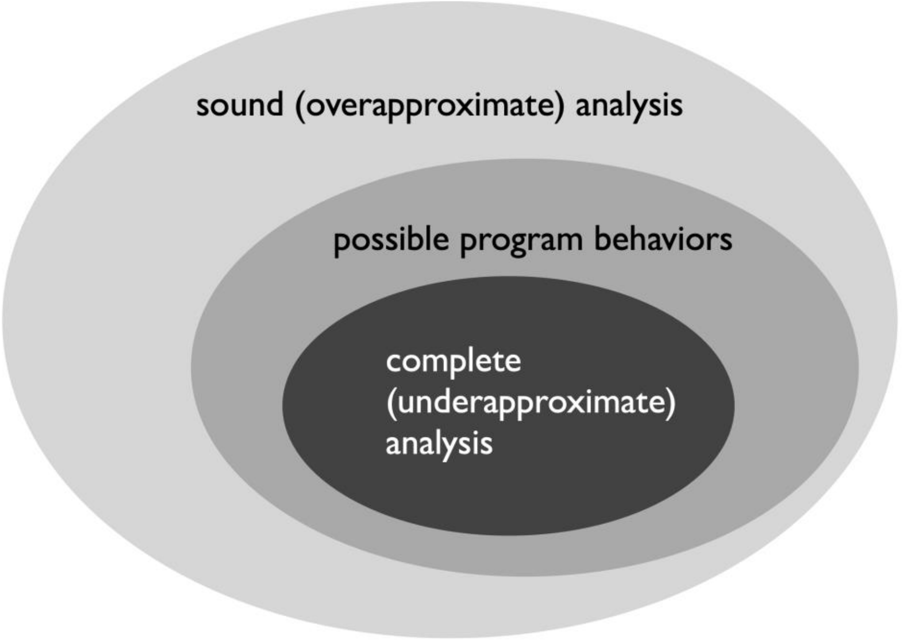
Can the “tainted” byte propagate to “A”?

```
if (A = 1)
{
   B = 2;
}
```

(Simplified): taint each input byte and see if the taint tag can be propagated to the condition

Soundness vs. Completeness

The fundamental design decision of almost all security analysis.

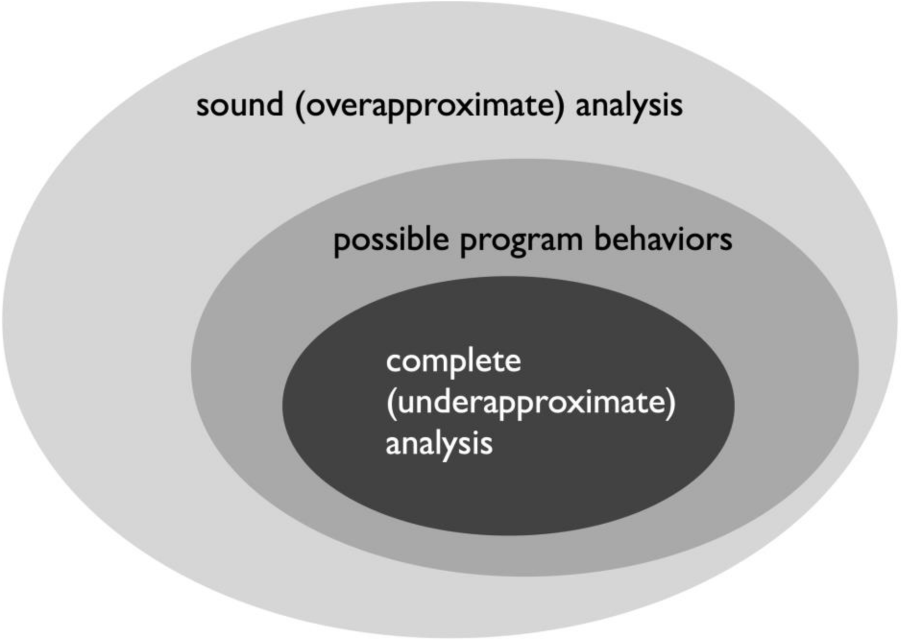


If a "sound" taint analysis tells you there is **no vulnerability** in your program, you know there is **no vulnerability**. → the *desired property of a security analysis*

- But if a "sound" analysis finds a vulnerability, it might be false positives (usually need to manually confirm)

Soundness vs. Completeness

The fundamental design decision of almost all security analysis.



If a "complete" taint analysis tells you there is a **vulnerability** in your program, you know it must be **a vulnerability**. → you don't need to manually confirm anyway

- But if a "complete" analysis finds no vulnerability, you will not be very happy…

Soundness vs. Completeness

The fundamental design decision of almost all security analysis.

How can I design a trivial sound analysis of buffer overflow?

- Treat very buffer access as "vulnerable". → lost every precision

How can I design a trivial complete analysis of buffer overflow?

- Treat very buffer access as "safe".

But they are just useless…

- *Design a "sound" and "useful" analysis is very very difficult…*
- *Abstract interpretation* provides a systematic framework to help you on that.
    - *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* POPL 1977

Likely give a Turning award to Cousot in the near future, we will see…

Taint analysis can be implemented as either sound or complete, or not *complete nor sound*.

- Unfortunately the last case is mostly what's happening in the real world..

Patrick Cousot