Intrusion Detection Via Static Analysis

IEEE Symposium on Security & Privacy 01’
David Wagner & Drew Dean

PRESENTED BY ZHENG SONG

David Wagner Drew Dean

Professor, UC Berkeley;

Security Research Group

SRI International Computer Science Laboratory

Princeton

Outline

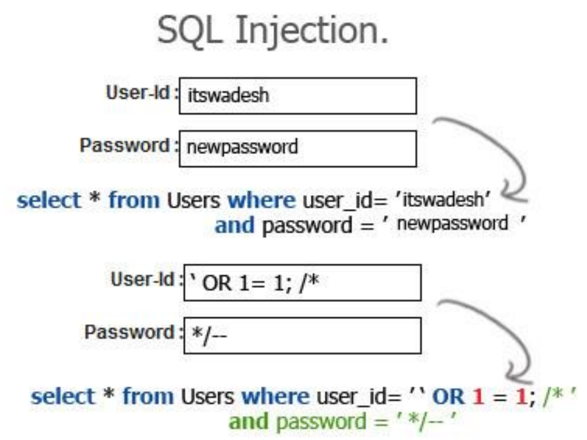
- Introduction
- Motivation
- Models
  - Trivial model
  - CallGraph model
  - Abstract stack model
  - Digraph model
- Implementation
- Evaluation

Introduction

1. Intrusion Detection System:

1) Examples: SQL Injection: what to do after?

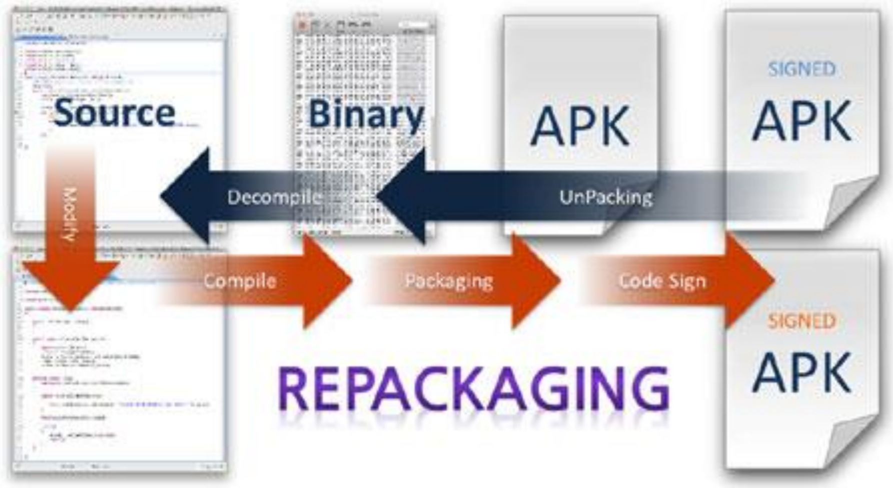
Android Repackaging Attack



SQL Injection.

```
User-Id : itswadesh
Password : newpassword
select * from Users where user_id= 'itswadesh'
                and password = ' newpassword '

User-Id : ' OR 1= 1; /*
Password : */--
select * from Users where user_id= '' OR 1 = 1; /* '
                and password = ' */-- '
```



2) Current Methods:

- Define a model of the normal behavior of a program; False alarm rate is too high!!! (false positive)
- List all the bad behaviors of a program….. False negative (0-day)

Motivation

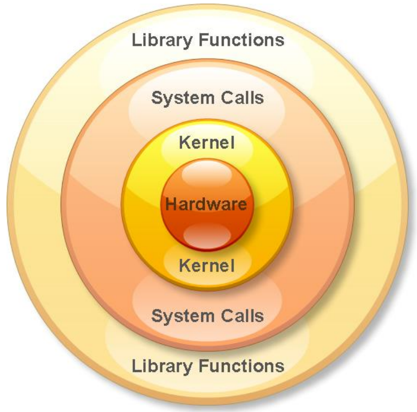
A intrusion detection system that:

1) wont cause too much false alarm;

2) can detect unknown attack;

Assumption: attack can only cause much harm when it interacts with the OS (system calls)

- System Calls: how a program requests a service from an operating system's kernel
- MySQL: open(); read(); close(); ->
  open(); read(); write(); close();
- Android: getGPS(); sendHttpRequest();Display();
  getGPS(); sendHTTPRequest();sendSMS();Display()



Plus One: solutions using static analysis?

1. find all reasonable system calls, when new system call pops out(no sequence) [trivial model]

Question: function calls -> system calls?

2. find the possible sequence of system calls, and detect non-existing paths.
   - Call graph model:
   - The abstract stack model:
   - The diagraph model:

Models:

1. A trivial model:

Create a set of system calls that the application can ever make

If a system call outside the set is executed, terminate the application

Pros: easy to implement

Cons: miss many attacks & too coarse-grained

Models:

2. Callgraph Model:

Build a control flow graph of the program by static analysis of its source or binary code

Result: non-deterministic finite-state automaton (NDFA) over the set of system calls

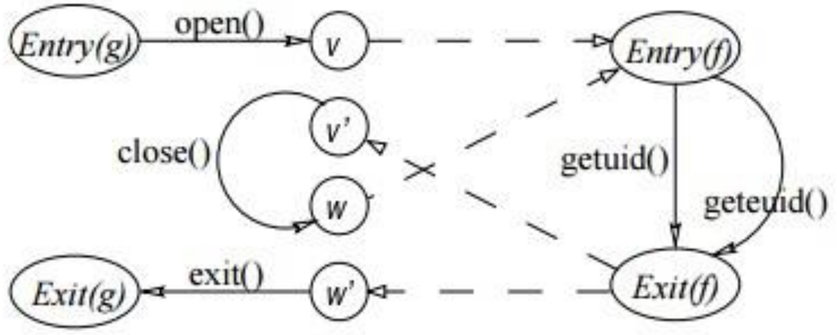
- Each vertex executes at most one system call
- Edges are system calls or empty transitions
- Implicit transition to special “Wrong” state for all system calls other than the ones in original code
- All other states are accepting

Call Graph Example

1. dash lines for inter-procedural edges;

2. how is it? Miss any attack? Cause false alarms? [f(1), close(fd), f(0)]   f(1)

```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```



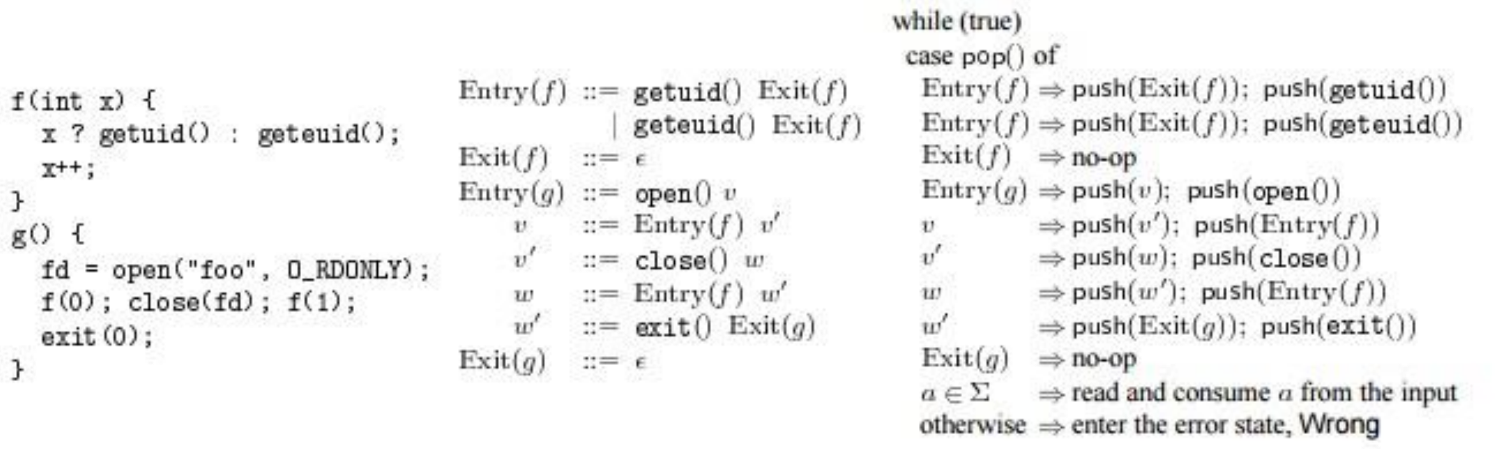
Models

3. Abstract Stack Model

Model application as a context-free language over the set of system calls:

- Build non-deterministic pushdown automaton (NDPDA)
- Each symbol on the NDPDA stack corresponds to single stack frame in the actual call stack
- All valid call sequences accepted by NDPDA; enter wrong state when an impossible call is made

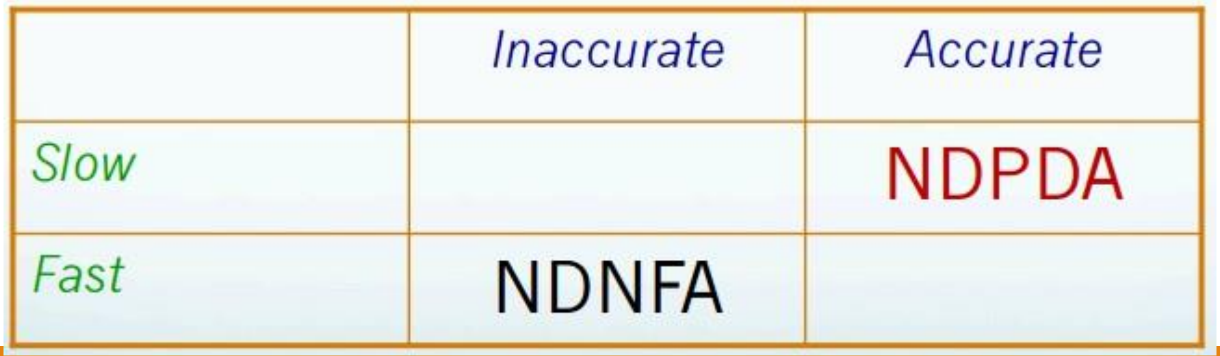
NDPDA Example



```
f(int x) {
  x ? getuid() : geteuid();
  x++;
}
g() {
  fd = open("foo", O_RDONLY);
  f(0); close(fd); f(1);
  exit(0);
}
```

$$
\begin{array}{lcl}
\text{Entry}(f) & ::= & \texttt{getuid()}\ \text{Exit}(f) \\
 & | & \texttt{geteuid()}\ \text{Exit}(f) \\
\text{Exit}(f) & ::= & \epsilon \\
\text{Entry}(g) & ::= & \texttt{open()}\ v \\
v & ::= & \text{Entry}(f)\ v' \\
v' & ::= & \texttt{close()}\ w \\
w & ::= & \text{Entry}(f)\ w' \\
w' & ::= & \texttt{exit()}\ \text{Exit}(g) \\
\text{Exit}(g) & ::= & \epsilon
\end{array}
$$

```
while (true)
 case pop() of
  Entry(f)   ⇒ push(Exit(f)); push(getuid())
  Entry(f)   ⇒ push(Exit(f)); push(geteuid())
  Exit(f)    ⇒ no-op
  Entry(g)   ⇒ push(v); push(open())
  v          ⇒ push(v'); push(Entry(f))
  v'         ⇒ push(w); push(close())
  w          ⇒ push(w'); push(Entry(f))
  w'         ⇒ push(Exit(g)); push(exit())
  Exit(g)    ⇒ no-op
  a ∈ Σ      ⇒ read and consume a from the input
  otherwise  ⇒ enter the error state, Wrong
```



	Inaccurate	*Accurate*
Slow		NDPDA
Fast	NDNFA	

Models

4. Digraph Model

Combines some of the advantages of the callgraph model in a simpler formulation

A list of possible k-sequences of consecutive system calls, starting at an arbitrary point

Pros: much more efficient than NDFA & NDPDA

Cons: less precise than NDFA & NDPDA

Optimizations:

Irrelevant systems calls

- Not monitoring harmless but frequently executed system calls such as brk()

System call arguments

Monitoring the arguments at runtime improves both precision and performance greatly

Evaluation

Time Consumption: obviously, digraph is the best; Precision:

The total height of the bar shows when arguments are ignored.

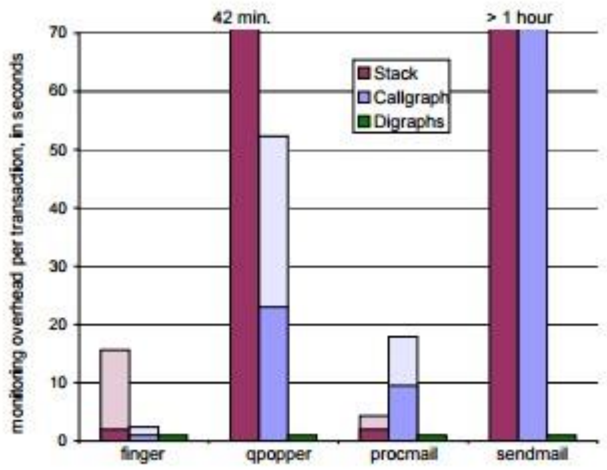


Figure 3. Overhead imposed by the run-time monitor for four representative applications, measured in seconds of extra computation per transaction.

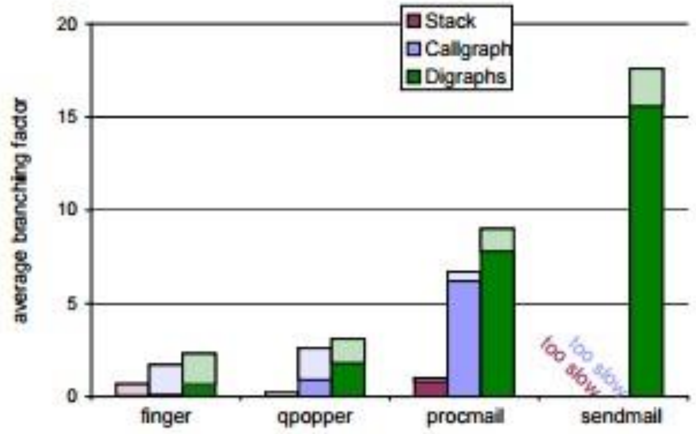


Figure 4. Precision of each of the models, as characterized by the average branching factor (defined later in Section 6). Small numbers represent better precision.

Unsolved Issues

Mimicry Attack

- Require high precision model to detect (poor performance)
- Runtime Overhead Use more advanced static analysis to get more precise models Later work such as VtPath, Dyck and VPStatic try to solve this problem

Models (discussion)

1. what is the benefits of using system calls instead of function calls?

2. How can we improve these methods?
   - Dynamic K according to the importance of system calls?
   - Parameters/ arguments?