Operating Systems - Advanced

Intrusion Detection via Static Analysis

David Wagner
U.C. Berkeley
daw@cs.berkeley.edu

Drew Dean
Xerox PARC
ddean@parc.xerox.com

Abstract

One of the primary challenges in intrusion detection is modeling typical application behavior, so that we can recognize attacks by their atypical effects without raising too many false alarms. We show how static analysis may be used to automatically derive a model of application behavior. The result is a host-based intrusion detection system with three advantages: a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms. We report on our experience with a prototype implementation of this technique.

Related work

- Most intrusion detection systems are based on one of two:
  1. a program model - analyze the behavior based on a set of known inputs
  2. a rules database
- The problem: inferring if the behavior is correct at runtime involves AI
  - AI is very imprecise
  - many false positives

Related work [2]

- Ko et al.: each program should be accompanied by the formal specification of the intended behaviour
- The “dream” of of the formal method community
- Impractical

Assumption

A compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions may be readily monitored.

Approach

1. Pre-compute a model of expected application behavior, built statically from program source code.
2. Monitor the program and check its system call trace for compliance to the model at runtime.

Result

- Host based intrusion detection system
  - semi-automated
  - protects against buffer overflows and other attacks that alter the normal execution paths (return to libc)
  - elimination of false alarms

Models

- The paper explores a few models:
  - a trivial one
  - the callgraph model
  - the abstract stack model
  - the digraphs model

The trivial model

- Let S be the set of system calls the application can ever make
- The set of allowable system call traces is then exactly the regular language S*

Callgraph model

- One problem with the trivial model is that it throws away the ordering of the system calls
- Build an NDFA over $\sum$ from the control flow graph of the program

Callgraph model [2]

- Build the CFG, G = {V, E}
- Assumes each node contains at most one system call
- The CFG is an NDFA with statespaceV U {Wrong}, transitions induced by E and alphabet ∑
- Wrong = a special state that is not accepting - used to identify the attack

Callgraph model [3]

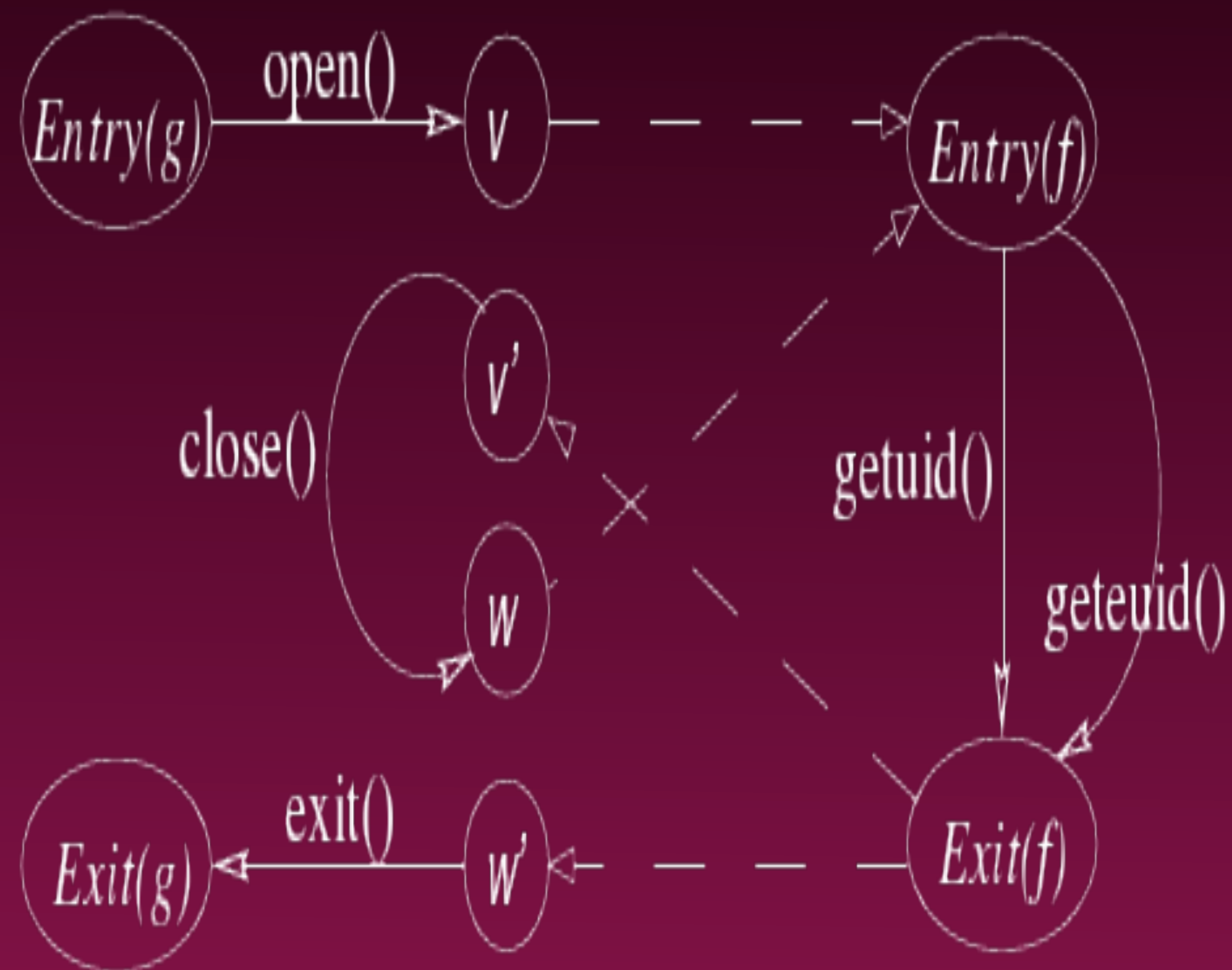
- Every edge $v \rightarrow w \in E$ from the CFG induces a transition in the NDFA::
- $v \xrightarrow{a} w$ if there is a system call “a” at the node “v”
- $v \xrightarrow{\epsilon} w$ for an empty transition, without a system call
- for every a∈∑ when the node v∈V doesn’t contain an outgoing transition induced by a, we add a transition $v \xrightarrow{a} \text{Wrong}$

Callgraph model [4]

- Every function call generates 4 special nodes: v, v', Entry(f) şi Exit(f)
- We then add two edges:
  - v -> Entry(f)
  - Exit(f) -> v'

Callgraph model [5]

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



Callgraph model [6]

- Simulate the operation of the NDFA on the observed system call traces
- Solve nondeterminism by exploring all possible paths

Callgraph model [6]

- This model cannot produce false alarms
- Imprecision because of impossible paths:
  - Example: v->Entry(f)->...->Exit(f)->w'
  - The attacks can exploit these paths

Abstract stack model

- Eliminates imposible paths
- Use a NDPDA to represent the model
- Equivalent to a context-free grammar
- Inputs will be system calls
- The NDPDA stack is an abstract version of the program stack

Example

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

The context-free grammar

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

NDPDA

$$
\begin{array}{l}
\text{while (true)} \\
\quad \text{case } \mathsf{pop}() \text{ of} \\
\qquad \begin{array}{ll}
\text{Entry}(f) & \Rightarrow \mathsf{push}(\text{Exit}(f));\ \mathsf{push}(\texttt{getuid}()) \\
\text{Entry}(f) & \Rightarrow \mathsf{push}(\text{Exit}(f));\ \mathsf{push}(\texttt{geteuid}()) \\
\text{Exit}(f) & \Rightarrow \text{no-op} \\
\text{Entry}(g) & \Rightarrow \mathsf{push}(v);\ \mathsf{push}(\texttt{open}()) \\
v & \Rightarrow \mathsf{push}(v');\ \mathsf{push}(\text{Entry}(f)) \\
v' & \Rightarrow \mathsf{push}(w);\ \mathsf{push}(\texttt{close}()) \\
w & \Rightarrow \mathsf{push}(w');\ \mathsf{push}(\text{Entry}(f)) \\
w' & \Rightarrow \mathsf{push}(\text{Exit}(g));\ \mathsf{push}(\texttt{exit}()) \\
\text{Exit}(g) & \Rightarrow \text{no-op} \\
a \in \Sigma & \Rightarrow \text{read and consume } a \text{ from the input} \\
\text{otherwise} & \Rightarrow \text{enter the error state, } \mathsf{Wrong}
\end{array}
\end{array}
$$

The monitoring algorithm

- Simulate the NDPDA with the intercepted system call traces
- Nondeterminism:
    - Exhaustive search: grows exponentially
    - Use better parsing algorithms (cannot use yacc because it doesn't accept nondeterminism)
- Need an efficient top-down parser
- Outside the scope of this paper

Digraph model

- Simple approach
- Accepts all k-sequences of consecutive system calls
- Authors implemented k=2, thus the name “digraph”

[offtopic: C trigraphs]

Trigraph	Equivalent
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

Building the model

- Start from the context free language of possible stack traces
- In order to see if $s \in \Sigma^k$ can occur during normal execution, the following condition needs to be met:
  - $(\Sigma^* s \Sigma^*) \cap L(\Gamma) \neq \emptyset$
- Complexity: $\Theta(k^3 \times |E| \times |\Sigma|^k)$

Monitoring algorithm

- Store last k-1 system calls, then check if the current system call completes a precalculated k-sequence
- Extremely scalable
- Less precise

Implementation problems

- Non-standard execution
  - pointers to functions, signals, setjmp/longjmp
- Libraries - static or dynamic
- Threads

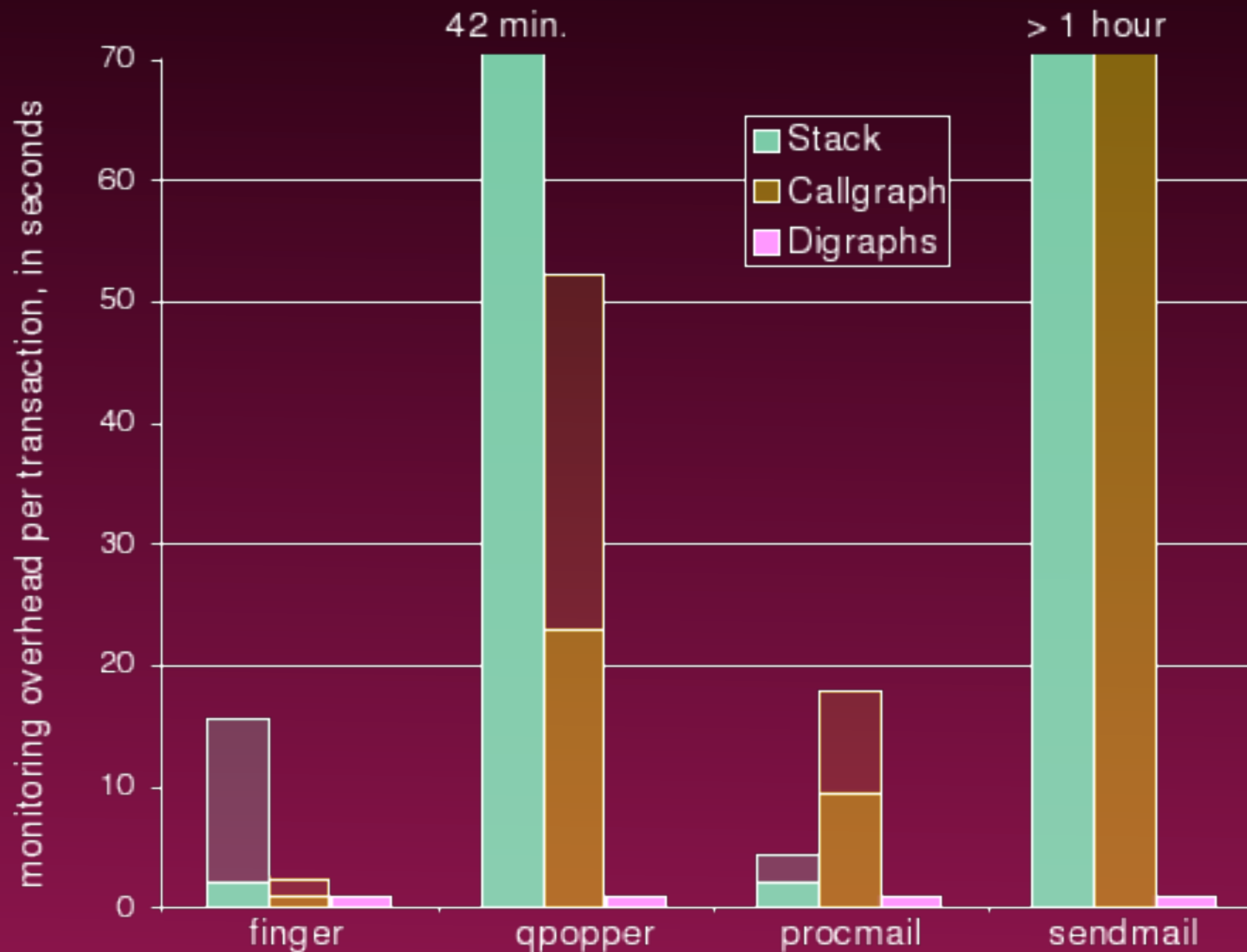
Solutions

- Pointers to functions: every pointer can refer any function
- Signals: enhance the CFG with pre and post guards for signal handling functions. Monitor signals and trigger transitions.
- Setjmp/longjmp: implemented in the runtime agent, rather than in the statical analysis
- Libraries: hand-crafted models
- Threads: kernel threads can be monitored, user threads impossible

Optimizări folosite

- Ignore irrelevant system calls
  - Example: brk
    - reduces the dimension of the automata - increases performance
    - can improve precision in the case of the digraph model
- Checking the system call arguments
  - recognize lexically constant system call arguments
  - increases performance by reducing ambiguities

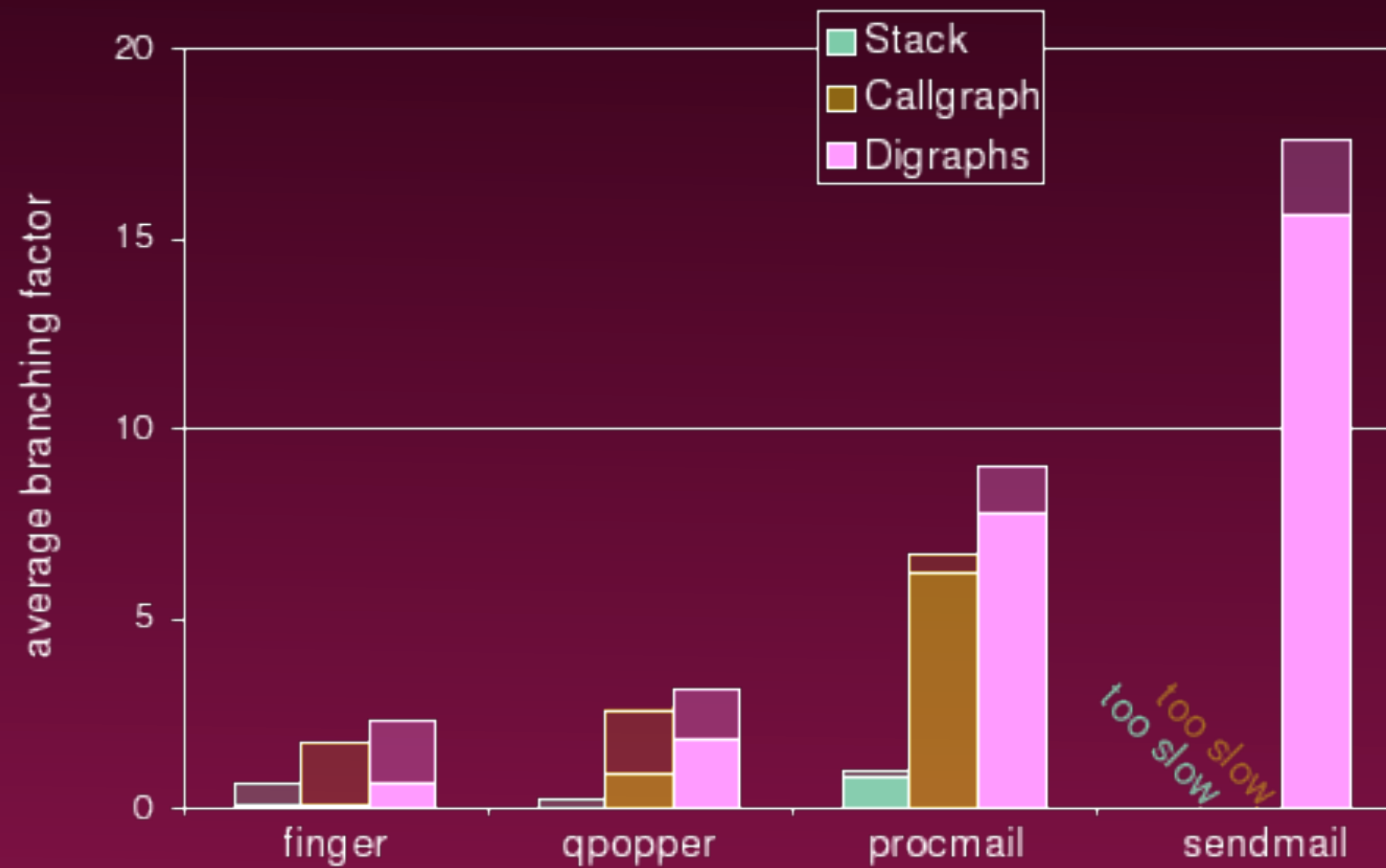
Performance evaluation



Performance evaluation [2]

- Transaction - one use case (e.g. sending a mail)
- Pentium II 450MHz, Linux, IBM JIT
- Java implementation

Precision evaluation



Precision evaluation [2]

- Branching factor = number of system calls that can be executed at any given time without triggering an alarm

Detected attacks

- Buffer overflows
- Root kits
- Even non-standard attacks:
  - E.g.: passing an environment variable to telnetd causes the dynamic linking with a library that belongs to the attacker