Semantic-based Intrusion Detection

Opportunities and Challenges

Jan Vitek

S3 Lab
Center for Education and Research in Information and Assurance and Security

PURDUE UNIVERSITY

Goals of this talk

- Intrusion detection techniques are essential to the protection of the national cyber-infrastructure
- Techniques that leverage programming language semantics can be used to automate intrusion prevention, detection, and response

There are many open problems. *We need you ...*

Agenda:

- Overview of the problem
- Describe state of the art
- Challenge problems

Scope of the Problem

- An intrusion results in an adversary being able to influence system behavior in ways unintended by it's designer or by it's user. Some sample attacks:
  - ☑ gain access to sensitive data stored on the compromised device
    could be addresses on a PDA
    passwords stored in clear text
  - ☑ inflict damage on the system
    cause slow bit-rot to gradually corrupt the filesystem
  - ☑ disrupt normal system operation
    consume bounded resources
  - ☑ mount distributed denial of service attacks
  - ...
- Intrusion detection systems aim to either catch intrusions in real-time (prevention) or identify intrusions by post-mortem data mining (assign blame)

Scope of the Problem (ii)

- Arguably, no intrusion had catastrophic consequences, how bad could it get?
- Supervisory Control and Data Acquisition (SCADA) systems

Intruders have hijacked an electric company's servers to host and play games...The intruders gained access to the power company's servers by exploiting a vulnerability in the company's file storage service, said NIPC, which would not name the power company. The intruders used the hacked FTP site to store and play interactive games that consumed 95% of bandwidth.This threatened the ability to conduct bulk power transactions. ... the incident seems just like a bunch of kids playing... they weren't even targeting the company...

-- www.merit.edu/mail.archives/netsec/2000-12/msg00042.html

In 2003, the Microsoft SQL Server worm—a.k.a.Slammer—infected a private computer network at the Davis-Besse nuclear plant, Ohio, disabling a safety monitoring system for 5 hours.Also, a process computer failed, and took about 6 hrs to become available again. Slammer also affected communications on the control networks of other electricity sector organizations by propagating so quickly that control system traffic was blocked.

-- www.gao.gov/htext/d04140t.html

Defining intrusions

An intrusion is a departure from normal system behavior
Dorothy Denning, Transactions on Sw. Eng., 13(2), 1987

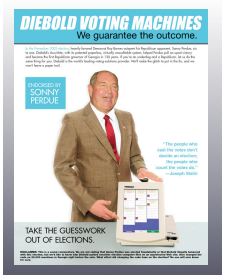
- Exploits are caused by faulty software. Faults which include:
  - ☑ programming errors
  - ☑ input validation errors
  - ☑ incorrect use of libraries
  - ☑ type errors
  - ☑ memory safety violations
  - ☑ intentional errors
- In all cases, *the behavior of the system diverges from its specification*. But since, specifications are more often than not either 1) missing, 2) incomplete, or 3) irrelevant to security; the definition is vacuous.

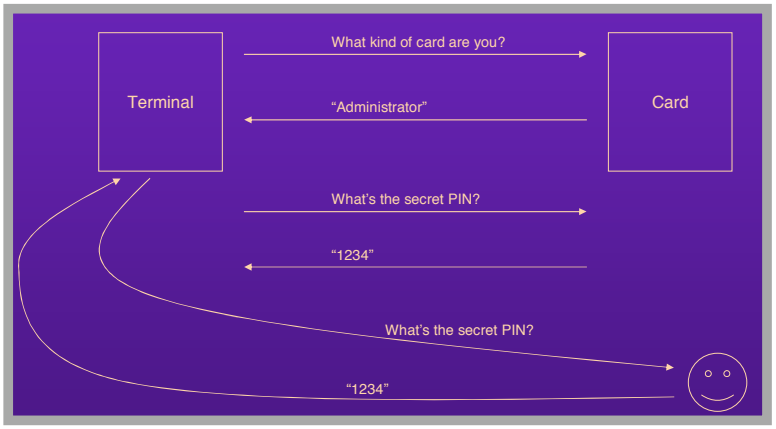
"I don't know how to define it, but I know it when I see it"
Supreme Court Justice Potter Stewart

Eradicating Intrusions

- Question: *can intrusions be avoided by making engineers aware of security issues?*
  Surely, highly-skilled professional trained to write secure software will write intrusion resilient codes?
- Answer: *No.* (and now for a digression)

Consider Dan Wallach's wonderful analysis of the Diebold voting software
www.ip3.gatech.edu/events/e-voting-risks-30minute-4up.pdf





Eradicating Intrusions

- Bottom line:

  Automated techniques for preventing and detecting flawed software will likely be needed.

  Just like a type system in a high-level language automated techniques can be used to rule out a set of simple mistakes

  Must go beyond legacy bugware

- wide-spectrum attacks

  a single kind of attack can affect a large number of software systems; responses are also widely applicable

- targeted attacks

  exploit the defect of one particular system; requires targeted response

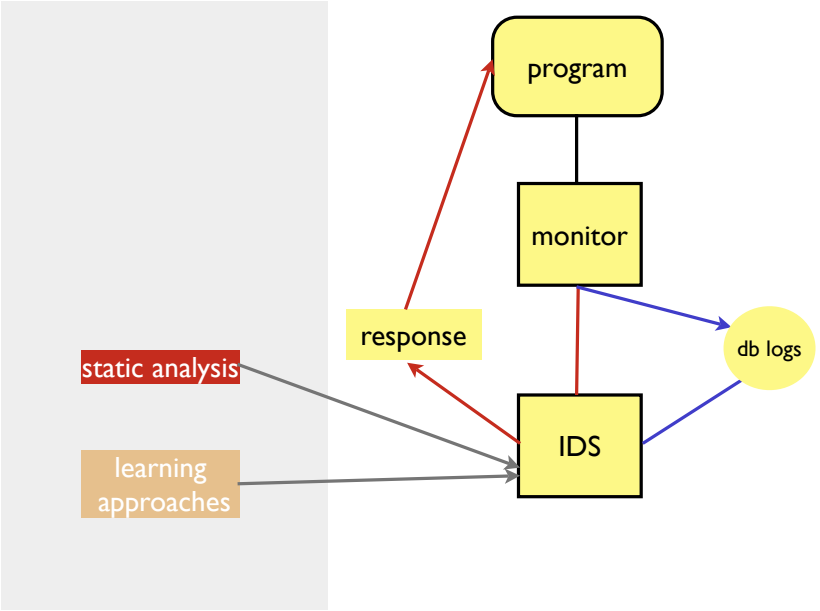
Formalizing Intrusions

- We define a program $\mathcal{P}$ as a state transition system $(S, \rightarrow, S_i)$.
- Given a set of initial state, S_i, we write $[\![\mathcal{P}]\!]$ to denote the set of all traces

$$[\![P]\!] \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \sigma_0 \in \mathcal{S}_\iota \wedge \forall i,\ \sigma_i \rightarrow \sigma_{i+1}\}$$

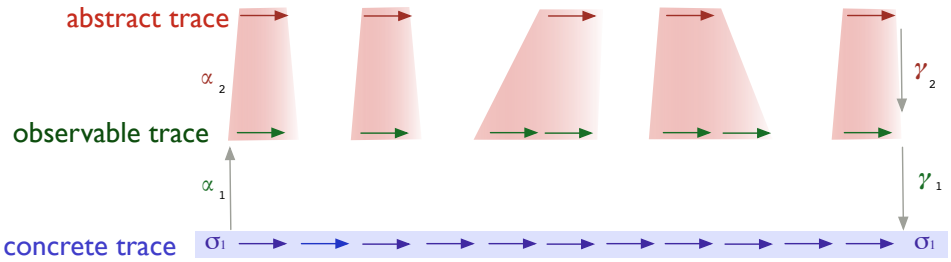
- For some set of traces $\mathcal{T}$, the predicate $good(\mathcal{T})$ denotes the set of valid traces
- $bad(\mathcal{T})$ denotes the set of attack traces. We have $good(\mathcal{T}) \cap bad(\mathcal{T}) = 0$
- An intrusion detection system will detect a set of attack traces, $\mathcal{A}(\mathcal{T})$
- A false negative is a trace $\sigma \in bad(\mathcal{T})$ - $\mathcal{A}(\mathcal{T})$
- A false positive is a trace $\sigma \in \mathcal{A}(\mathcal{T}) \cap good(\mathcal{T})$
- *The cost of a false positive is measured in lost user confidence and system administrator involvement*
- *Take a false positive rate of 1%, and assume that an attack occurs once per 100000 runs then we have 1000 false warnings per attack*

IDS Overview



Semantics-based IDS

- Traces are unwieldy - *too detailed, difficult to monitor in real-time or store*
- Program traces are abstracted to observable traces, $observe([\![\mathcal{P}]\!])$
- An IDS define a finite characterization of acceptable traces, as an abstract interpretation over observable traces



- Can $observe(good([\![\mathcal{P}]\!])) \cap observe(bad([\![\mathcal{P}]\!])) \neq 0$?
- If yes, the notion of observation is too weak

IDS through Static Analysis

"Intrusion Detection via Static Analysis", Wagner and Dean, Security&Privacy, 2001

- Use of static program analysis to obtain the control flow graph of a program. The CFG is viewed as a specification -- traces that can not occur in the CFG are deemed invalid. The approach guarantees that there are no false positives.
- Observables are operating system calls; monitoring is done by interposition
- The key challenge is to increase precision (fewer false negatives) while ensuring that the monitoring overheads are reasonable.
- 3 models of program behavior NFA, PDA, and k-bounded strings

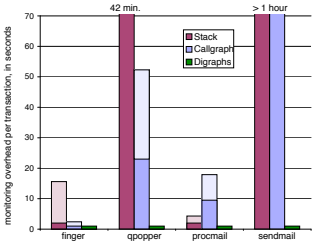


Figure 3. Overhead imposed by the run-time monitor for four representative applications, measured in seconds of extra computation per transaction.

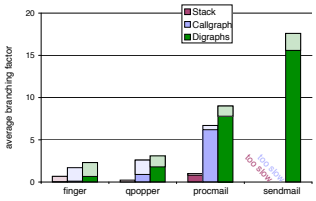
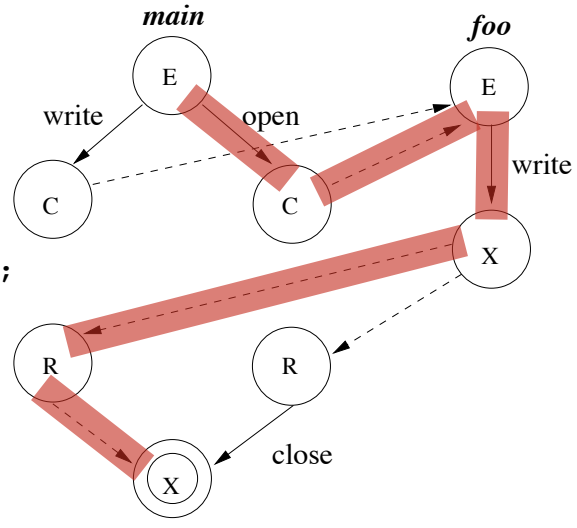


Figure 4. Precision of each of the models, as characterized by the average branching factor (defined later in Section 6). Small numbers represent better precision.

Example - NFA

```
main( int argc, char** argv) {
   int fd;
   if ( argc == 1 ) {
     write(1, "StdOut", 6);
     foo(1);
   } else {
     fd = open(argv[1], O_WRONLY);
     foo(fd); close(fd);
}  }
void foo(int x) {
   write(x,"Hello World",11);
}
```



The NFA representation of the program includes trivially infeasible paths.
To avoid those it is necessary to retain calling context information.

Inlined Automaton Model

"Efficient Intrusion Detection using Automaton Inlining", Gopalakrishna. Spafford, Vitek, Security&Privacy, 2005

- The cost of monitoring can be reduced to be < 10%

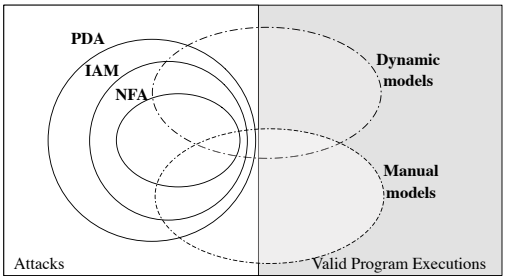
Program	Unmonitored	Base	Monitored	%
cat	58.95	59.14	59.04	0
htzipd	13.22	13.35	14.46	8.3
lhttpd	29.17	28.82	29.08	0.9
gnatsd	35.61	36.29	34.36	0

Table 4. Program runtime in seconds.

	% Runtime Overhead			% Memory Overhead		
Program	Dyck	VPStatic	IAM	Dyck	VPStatic	IAM
cat	56	32	0	49	194	0.2
htzipd	135	97	8.3	38	183	7.1

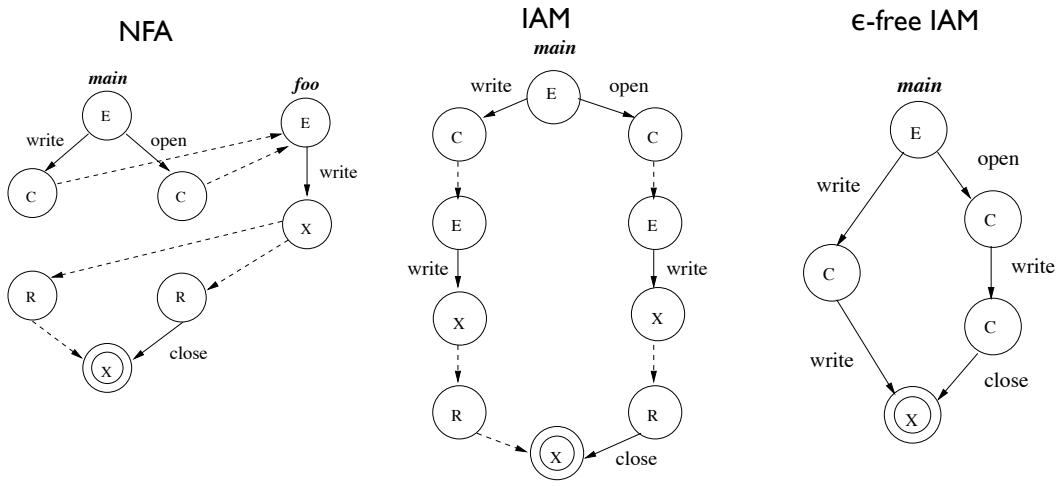
Table 6. Comparing models.

- observation: for real-time monitoring, trading space for speed is acceptable
- goal is to obtain a model of behavior as precise as PDAs and as efficient as NFAs



Building an IAM

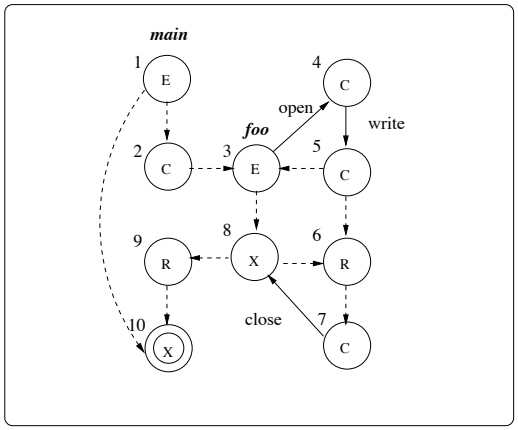
- The IAM is built from the NFA by inlining all functions. The ϵ-free IAM is then built by deleting all non-observable transitions. Standard automaton minimization techniques can also be applied



Dealing with Recursion

- Recursive calls are not inlined, and can thus lead to infeasible paths in the model
- These only affect precision if there are observable actions in the unwinding path

```
main( int argc, char** argv) {
   if (argc > 1) foo(--argc, argv);
}
void foo(int argc, char **file) {
   int fd;
   if ( argc != 0 ) {
     fd = open(file[argc], O_WRONLY);
     write(fd,"Hello World",11);
     foo(--argc, file); close(fd);
  }
}
```



Mimicry Attacks

Mimicry Attacks on Host-Based Intrusion Detection Systems, Wagner and Soto, CCS'02

- A mimicry attack occurs when the attacker takes advantage of

$$observe(good([\![\mathcal{P}]\!])) \cap observe(bad([\![\mathcal{P}]\!])) \neq 0$$

- If the attacker understands the IDS well enough, then it is possible to craft a valid sequence of observable events which have the attack as a subtrace

```
read() write() close() munmap() sigprocmask() wait4()
sigprocmask() sigaction() alarm() time() stat() read()
alarm() sigprocmask() setreuid() fstat() getpid()
time() write() time() getpid() sigaction() socketcall()
sigaction() close() flock() getpid() lseek() read()
kill() lseek() flock() sigaction() alarm() time()
stat() write() open() fstat() mmap() read() open()
fstat() mmap() read() close() munmap() brk() fcntl()
setregid() open() fcntl() chroot() chdir() setreuid()
lstat() lstat() lstat() lstat() open() fcntl() fstat()
lseek() getdents() fcntl() fstat() lseek() getdents()
close() write() time() open() fstat() mmap() read()
close() munmap() brk() fcntl() setregid() open() fcntl()
chroot() chdir() setreuid() lstat() lstat() lstat()
lstat() open() fcntl() brk() fstat() lseek() getdents()
lseek() getdents() time() stat() write() time() open()
getpid() sigaction() socketcall() sigaction() umask()
sigaction() alarm() time() stat() read() alarm()
getrlimit() pipe() fork() fcntl() fstat() mmap() lseek()
close() brk() time() getpid() sigaction() socketcall()
sigaction() chdir() sigaction() sigaction() write()
munmap() munmap() munmap() exit()
```

- To counter mimicry attack, the set of observables must be extended
  - examples include function arguments, and other parts of the internal state
  - better static analysis algorithm can reduce the set of admissible paths

Self-Monitoring with Guards

Protecting Software Code by Guards, Hoi Chang and Mike Atallah, Workshop on Security&Privacy in Digital Rights Management, 2001

ARXAN

- The idea is simple: embedded guards into the code by binary rewriting
- Guards are designed so as to be difficult to circumvent
- The behavior of the guard is interspersed with the program code itself (mimicry for a good cause)
- Guards can be used to add observable action to a code, they can implement arbitrary predicates over the state of the system

Observing Causality

- The choice of what to observe often depends on some notion of causality
- For example, when calling `exec(com, s, a)` does the string `com` originate from an input to the program?
- In the language perl there is a dynamic taint mode which tracks string construction operations such that any string built out of substrings coming from a program input is marked as tainted
- Of course this is an approximation that is not sound due to indirect flows, and a more precise notion can not be tracked dynamically -- something like information flow is needed

Secure Composition of Untrusted Code: Box π, Wrappers, and Causality Types, Sewell and Vitek, Journal of Computer Security 2001

VMM

A Virtual Machine Introspection Based Architecture for Intrusion Detection, Garfinkel and Rosenblum, Network and Distributed System Security 2003.

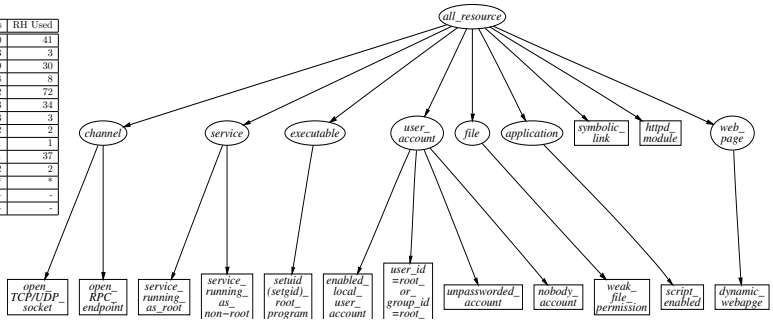
- System call interposition techniques are limited to monitoring calls and their payload; the responses are typically limited to termination
- Virtual Machines such as VMWare or Xen virtualize the operating system and can do much more
- An IDS integrated in the VMM can observe and control resource usage of a process
- The state of the process can be introspected and, for example, scheduling can be modified (e.g. slow down a process that looks suspicious)

Metrics

- Metrics are needed to measure the vulnerability of software systems and the quality of defenses
- Some early work, but more needed

Attack Class	Debian	RH Default	RH Facilities	RH Used
open_TCP/UDP_socket	15	12	40	41
open_remote_procedure_call(RPC)_endpoint	3	3	3	3
service_running_as_root	21	26	29	30
service_running_as_non-root	3	6	8	8
setuid(setgid)_root_program	54	54	72	72
enabled_local_user_account	21	25	33	34
user_id=root_or_group_id=root_account	0	4	3	3
unpassworded_account	0	0	2	2
nobody_account	1	1	1	1
weak_file_permission	7	7	21	37
script_enabled	1	2	2	2
symbolic_link	*	*	*	*
httpd_module	-	-	-	-
dynamic_web_page	-	-	-	-

Table 1: Attack surface measurement results



Measuring a System's Attack Surface, Manadhata and Wing, CMU 2004

Binary Analysis

- Binaries are often the only artifact at hand & they are language independent, thus algorithms that work on binaries are widely applicable
- The first challenge is to extract a control flow graph from the binary (a form of disassembly)
- Executables can be obfuscated (IP protection), stripped, optimized...
- Tools are needed to analyze binaries either statically or on-the-fly

Static Disassembly of Obfuscated Binaries, Kruegel, Robertson, Valeur and Vigna, UC Santa Barbara

Beyond Monitoring

- *Pathological fan-out*

  Usefulness of automatic model extraction is limited by the underlying software system. Consider a program which consists of one big switch statement, and on each branch of the switch a different system call. Monitoring is ineffective because the program can legally generate all sequence of systems calls. (essentially this is an interpreter)
- *Unpredictable control flow*

  programs implementing their own threading or using complex function pointer arithmetic cannot be monitored
- *Dynamic code generation*

  code that was not available at model generation time can not be monitored. (dynamic model generation?)

Conclusions

- Semantics and Static Analysis have an important role to play in software security
- Many open problems require urgent attention
- No silver bullet, but we are raising the bar