CS5231: Systems Security

Lecture 9: Audit Applications

Main Applications (Continue from Week 8)

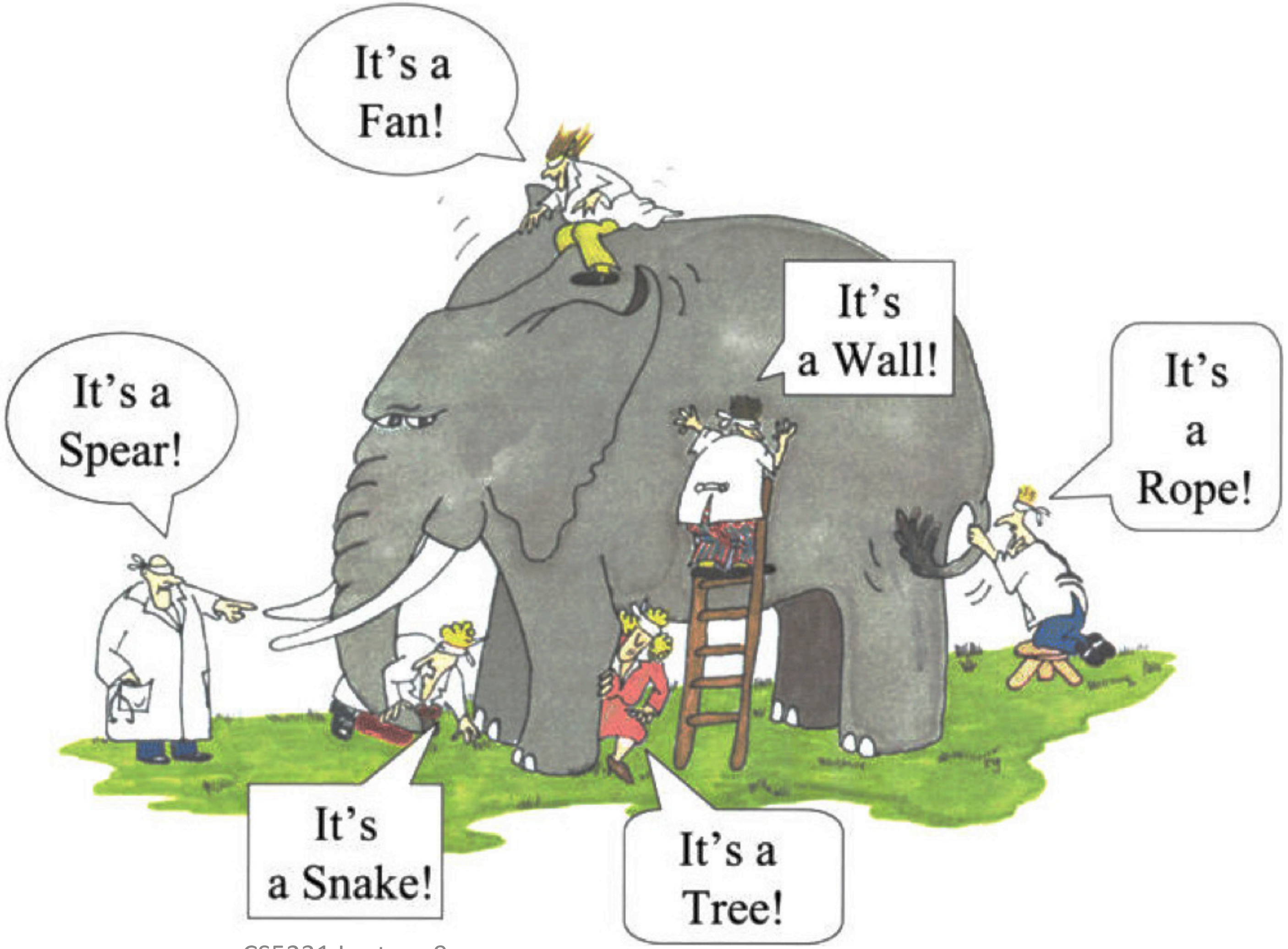
- Logging-based Applications
  - Intrusion Detection
  - Intrusion Recovery
  - Software Debugging
- One question to think: What to log?
  - Depends on applications
  - Need the right abstraction and amount of information

Levels of Understanding of Cyber Security Events

Transaction Level

System-call/Audit Level

Program/Instruction Level



Binary-Level View of an Incident

Assembly code

```
myfunc:
push    {fp, lr}
add     fp, sp, #4
ldr     r0, helloworld
bl      <puts>
mov     r3, #0
mov     r0, r3
pop     {fp, pc}
```

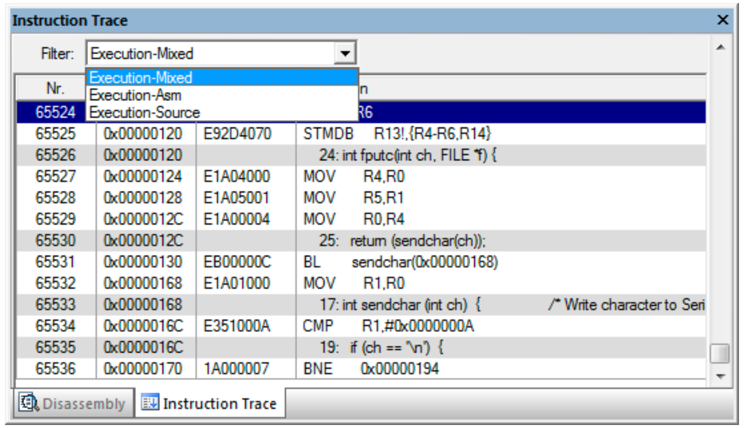
Source code

```
void myfunc (){
  printf("hello world");
}
```

Binary code

```
00 48 2d e9 04 b0 8d e2
0c 00 9f e5 a5 ff ff eb
00 30 a0 e3 03 00 a0 e1
00 88 bd e8 d0 04 01 00
```

Instruction Trace



Control-flow Graph

Audit-Log-Level View

- **User-space** utilities (e.g., Auditd) collect system call records from kernel space through Netlink and write them to a log file under `/var/log/audit`
  - An Example of a read log entry in Auditd

```
----
type=PROCTITLE msg=audit(15/08/2019 14:37:30.522:61916019) : proctitle=sshd: junzeng [priv]
type=SYSCALL msg=audit(15/08/2019 14:37:30.522:61916019) : arch=x86_64 syscall=read
success=yes exit=52 a0=0x3 a1=0x7ffd69eecad0 a2=0x4000 a3=0x7ffd69ef0a60 items=0 ppid=5512
pid=5542 auid=junzeng uid=junzeng gid=junzeng euid=junzeng suid=junzeng fsuid=junzeng
egid=junzeng sgid=junzeng fsgid=junzeng ses=1805 comm=sshd exe=/usr/sbin/sshd key=(null)
----
```

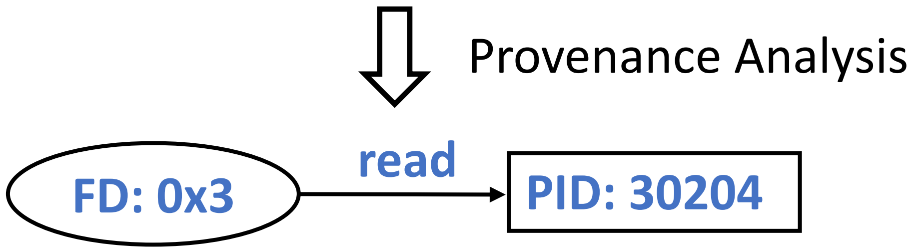
  - An Example of a read log entry in Auditbeat

```
{"@timestamp":"2020-11-04T14:27:14.666Z","@metadata":{"beat":"auditbeat","type":"doc",
"version":"6.8.12"},"auditd":{"sequence":989996,"result":"success","session":"1402","data":
{"a3":"20656c706f657020","tty":"(none)","a2":"1000","arch":"x86_64","syscall":"read",
"exit":"4096","a1":"5583baa77f70","a0":"5"}},"user":{"name_map":{"suid":"root",
"auid":"junzeng","egid":"root","euid":"root","fsuid":"root","gid":"root","sgid":"junzeng",
"fsgid":"root","uid":"root"},"euid":"0","fsgid":"0","fsuid":"0","suid":"0","gid":"0",
"sgid":"1000","egid":"0","auid":"1000","uid":"0"},"process":{"exe":"/usr/sbin/sshd",
"pid":"7959","ppid":"1689","name":"sshd"}}
```

Building Dependency Graph

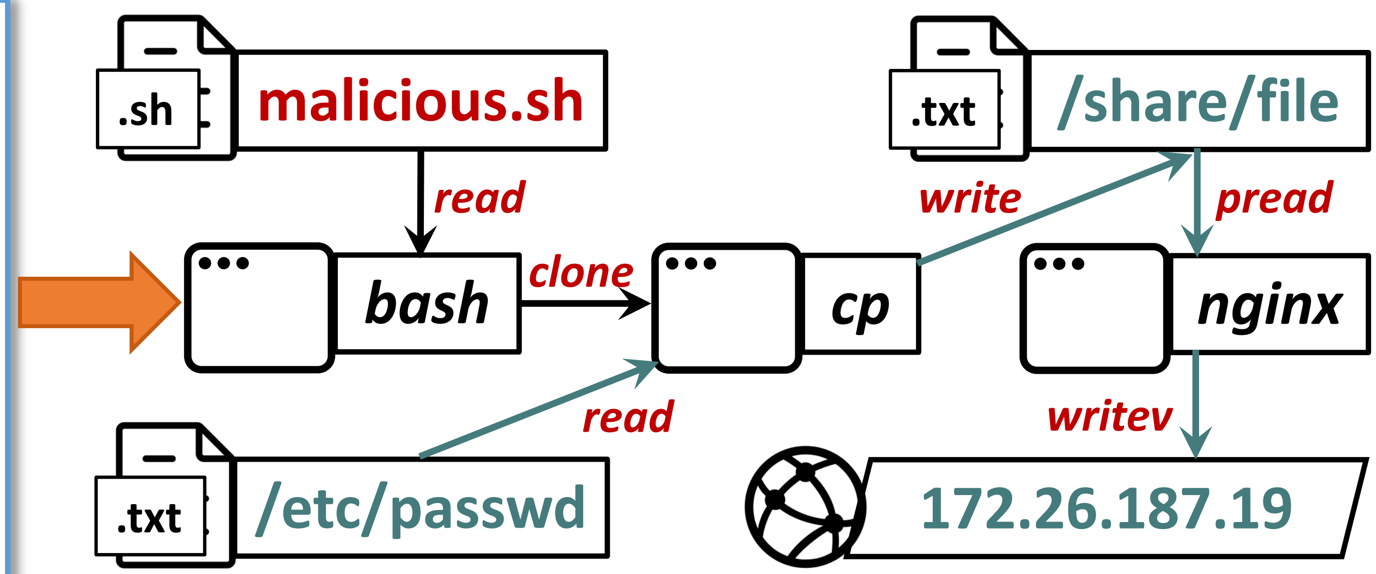
- Nodes
  - Files, processes, sockets
- Edges
  - System calls

type=SYSCALL msg=audit(**30/09/19 20:34:53**.383:98866813) : arch=x86_64 **syscall=read** exit=25 **a0=0x3** ppid=15757 **pid=30204** auid=junzeng sess=6309



Provenance Graph: a representation of audit logs

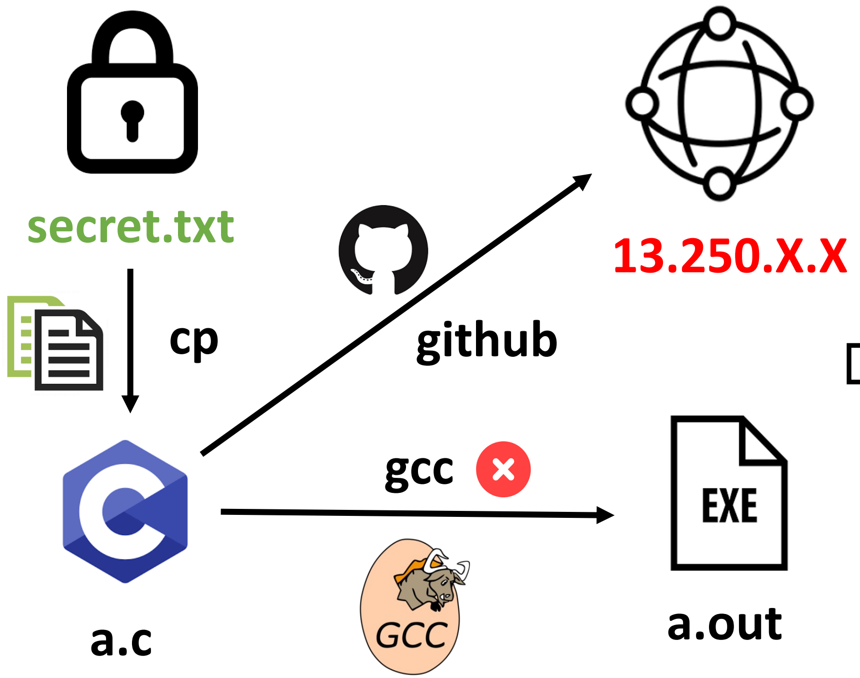
```
…
1. bash, read, malicious.sh
2. bash, clone, cp
3. cp, read, /etc/passwd
4. cp, write, /share/file
5. nginx, pread, /share/file
6. nginx, writev, 172.26.187.19
…
```



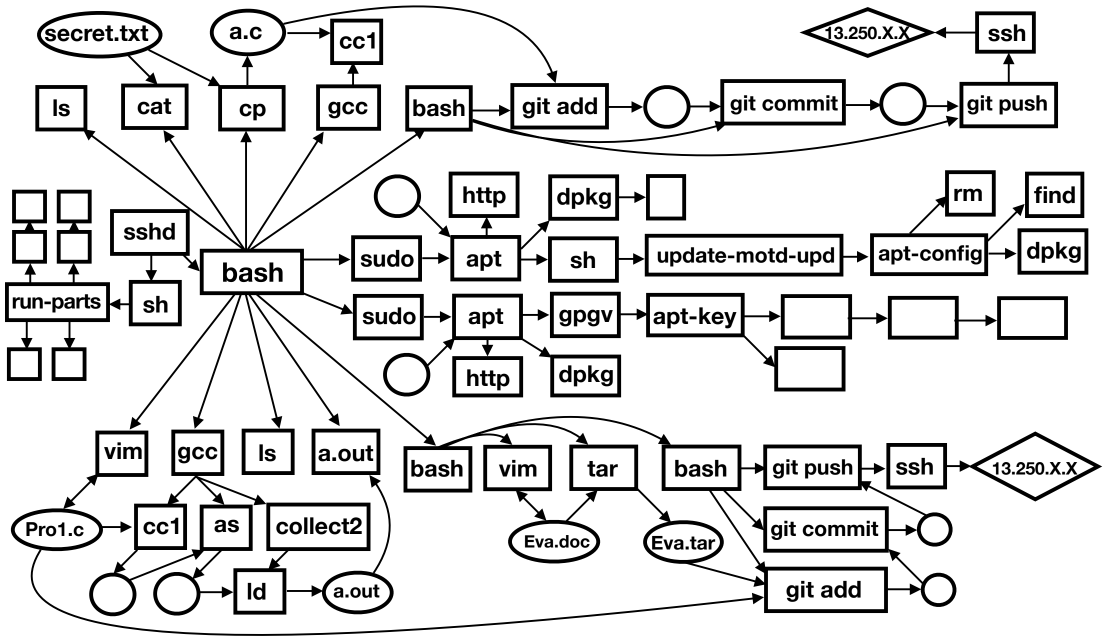
✓***Provenance Graph*** constructs the ***overall attack scenario*** by ***combining*** historic audit logs***!***

An Example

Attack Scenario: A software tester exfiltrates sensitive data that he has access to



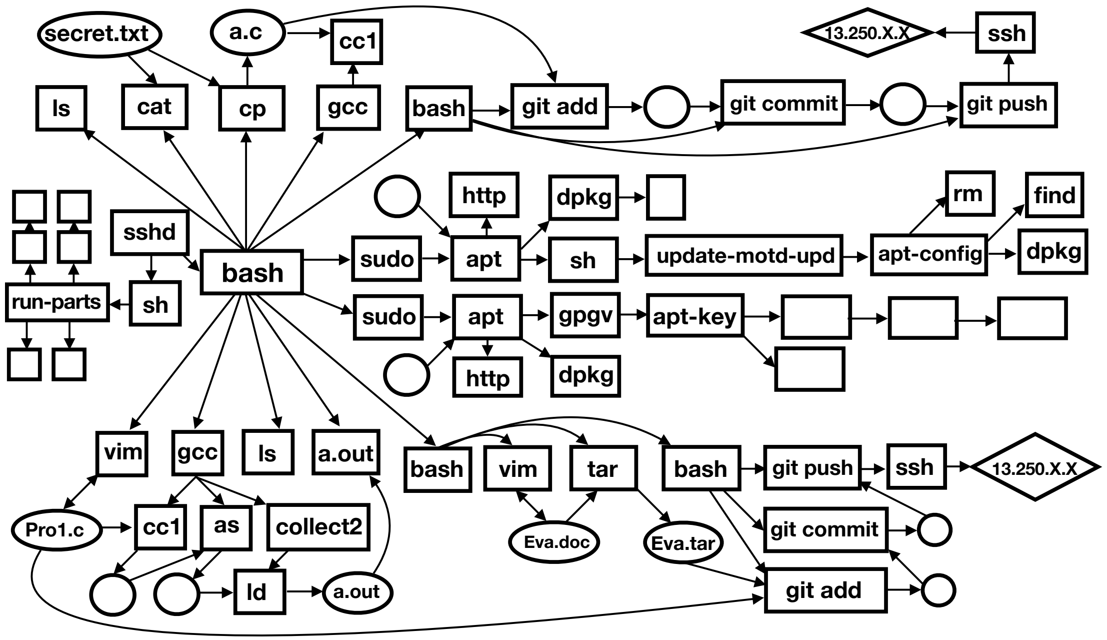
Data Exfiltration Steps



Provenance Graph

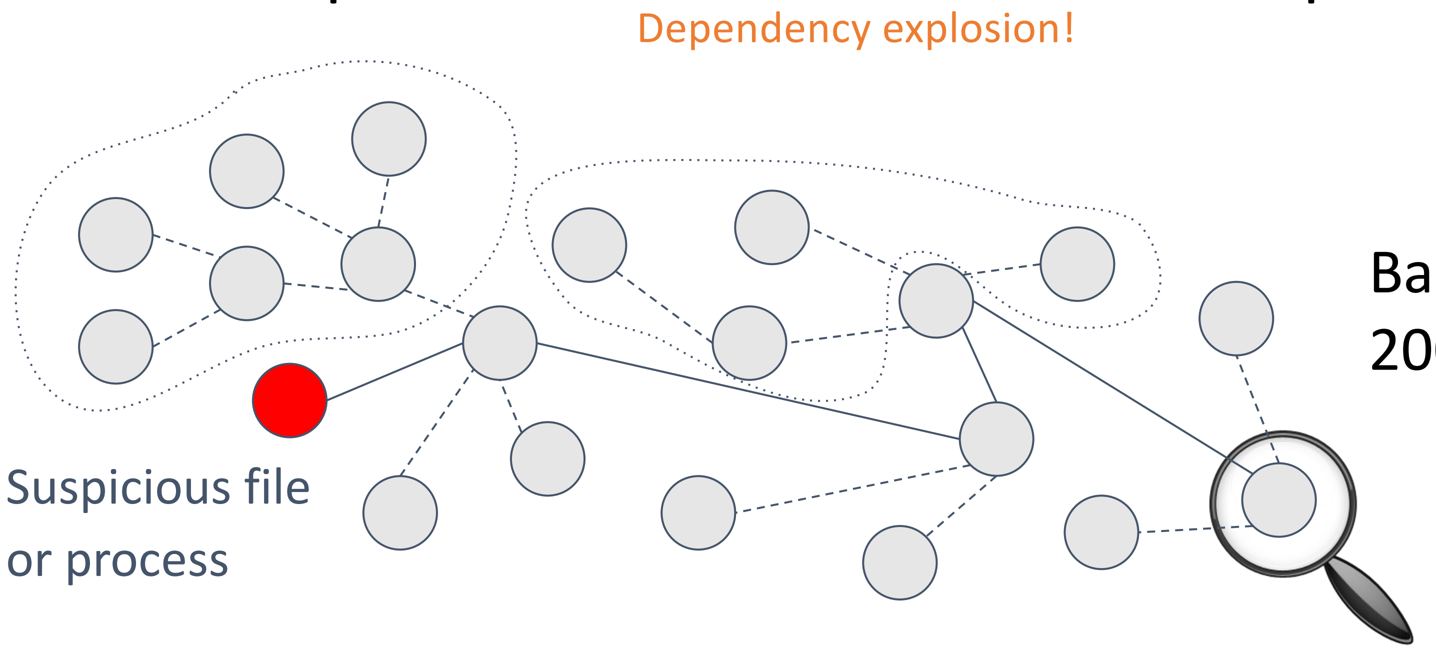
Analysis of Provenance Graph

- Dependency analysis
- Subgraph matching
- Deep learning and recommendation



Dependency Analysis

- Starting from a detection point, *Backtracker* does:
  - Events & objects identification related detection point
  - Generate dependency graph
  - Use rules to prune unrelated nodes in the dependency graph



Backtracker (King & Chen, 2003)

Intrusion Detection

- Intrusion Detection is the process of identifying and responding to malicious activity targeted at computing and networking resources
- Resources:
  - One computer, or
  - A local/wide area network

Models of Intrusion Detection

- Anomaly detection
  - What is usual, is known
  - What is unusual, is bad
- Misuse detection
  - What is bad, is known
  - What is not bad, is good
- Specification-based detection
  - What is good, is known
  - What is not good, is bad
- Goal ➔ generating a **Detection Point**

Detection Point

- Suggests a possible intrusion
- Examples:
  - An anomaly log entry
    - e.g., a shell process launched
  - A suspicious system activity
    - e.g., an outbound TCP connection to a remote IRC server
  - An unauthorized modification to a critical configuration file
    - e.g., /etc/inetd.conf

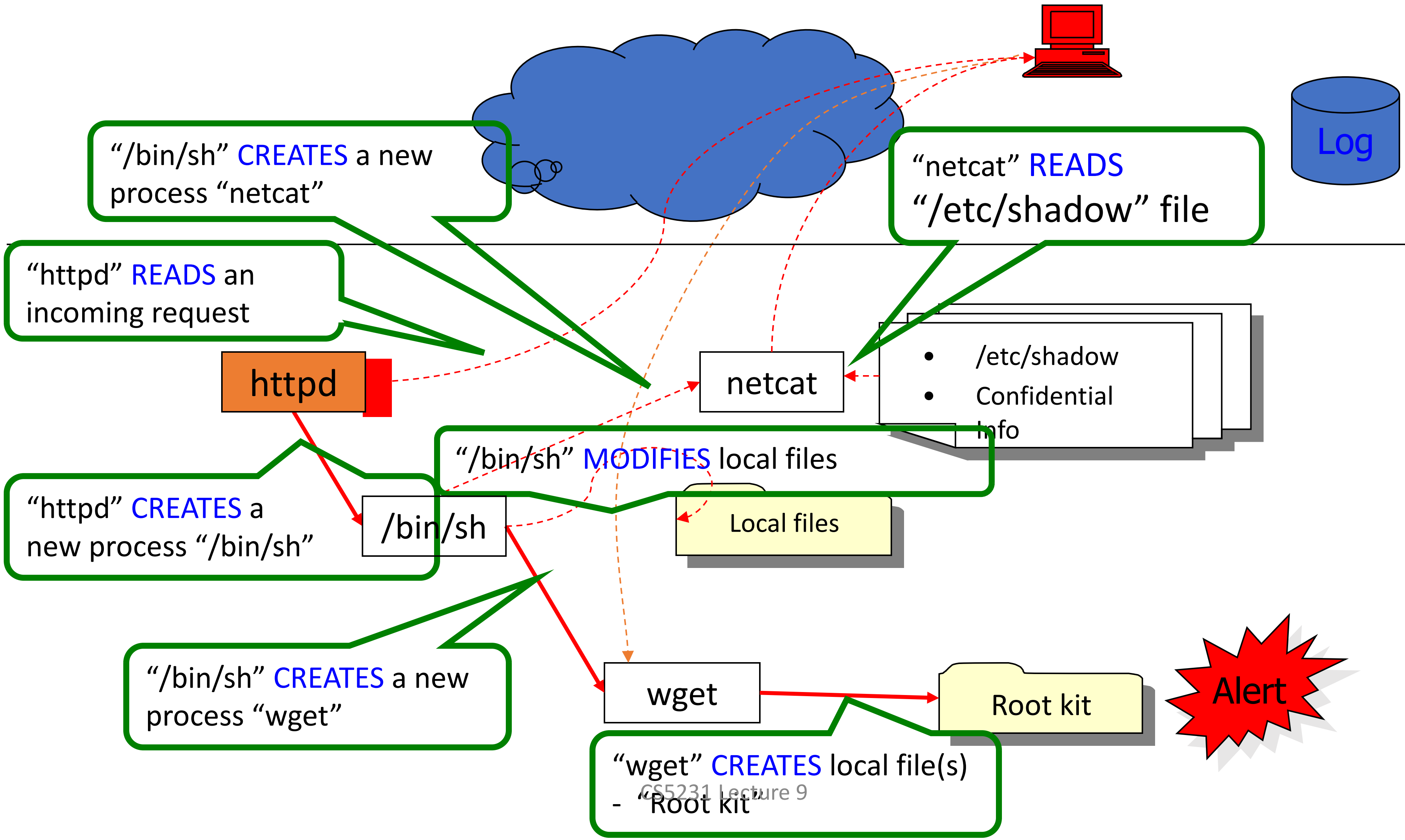
After an Intrusion Is Identified

- For each intrusion, it is desirable to find out:
  - **Break-in Point:**
    - How did the attacker gain access to the system?
  - **Contaminations:**
    - What did the attacker do after the break-in?

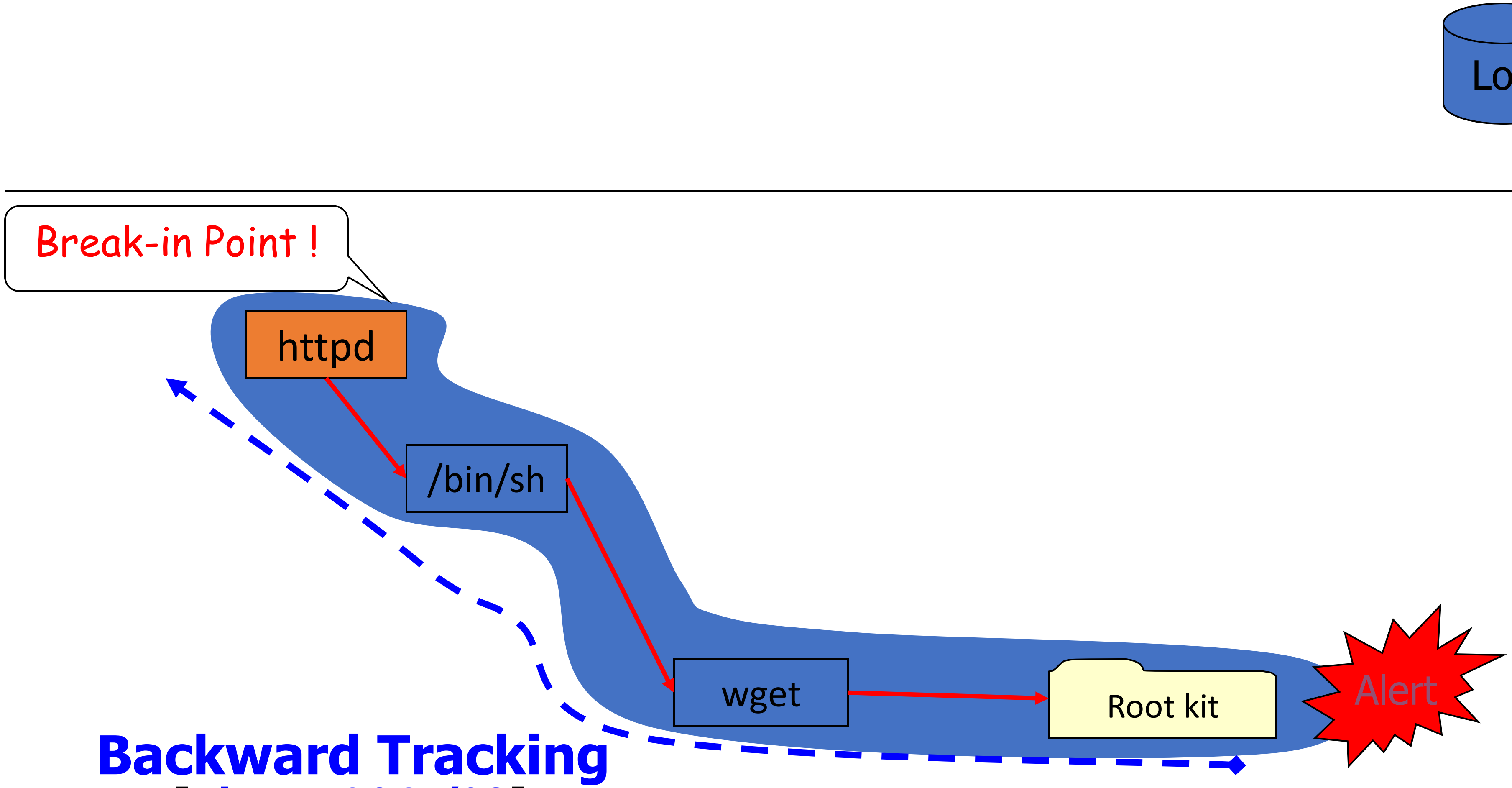
Intrusion Investigation

- Three Main Steps
  - Step 1: Online Log Collection
  - Step 2: Backward Tracking
  - Step 3: Forward Tracking

Step 1: Online Log Collection

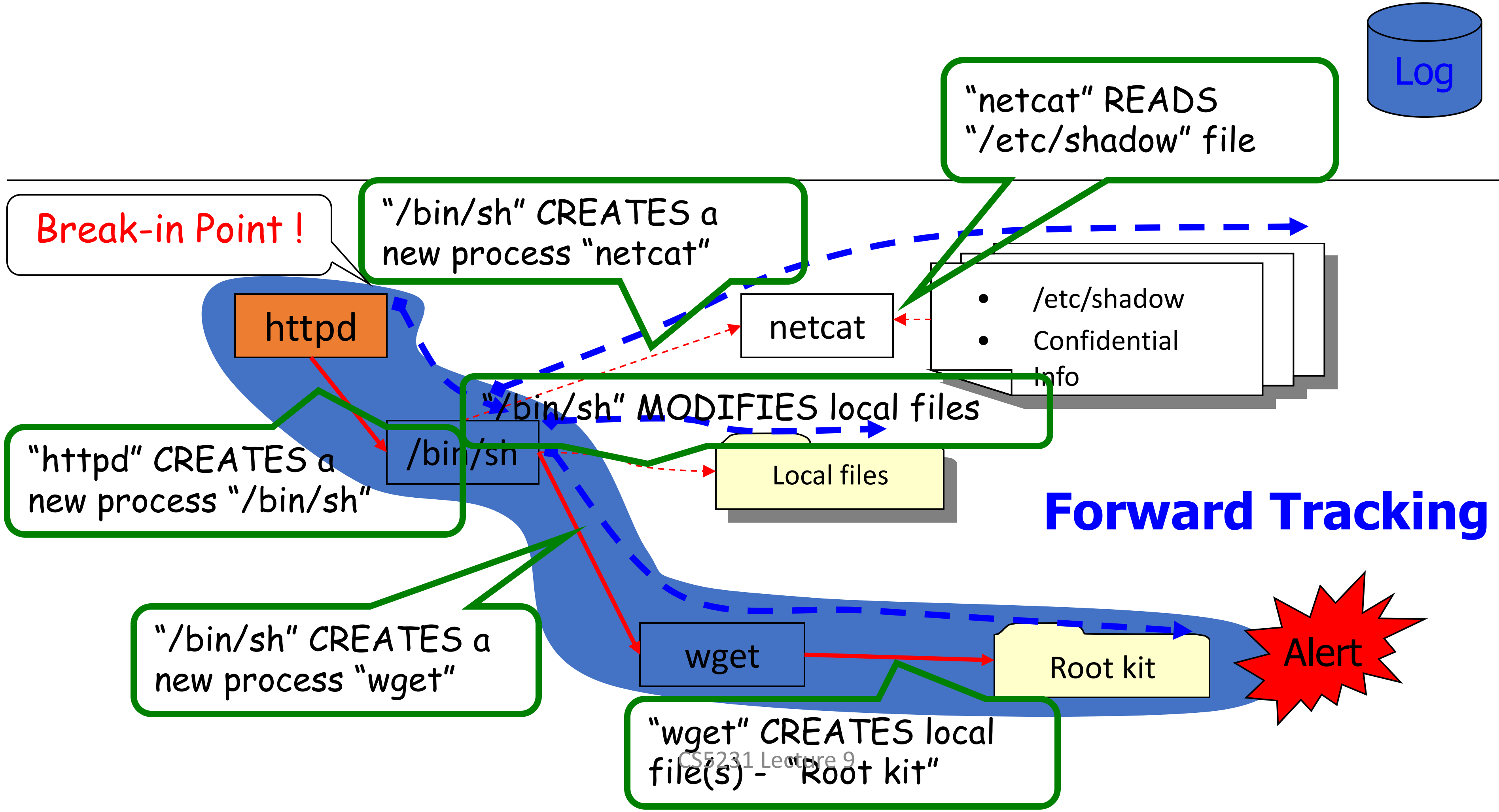


Step 2: Backward Tracking

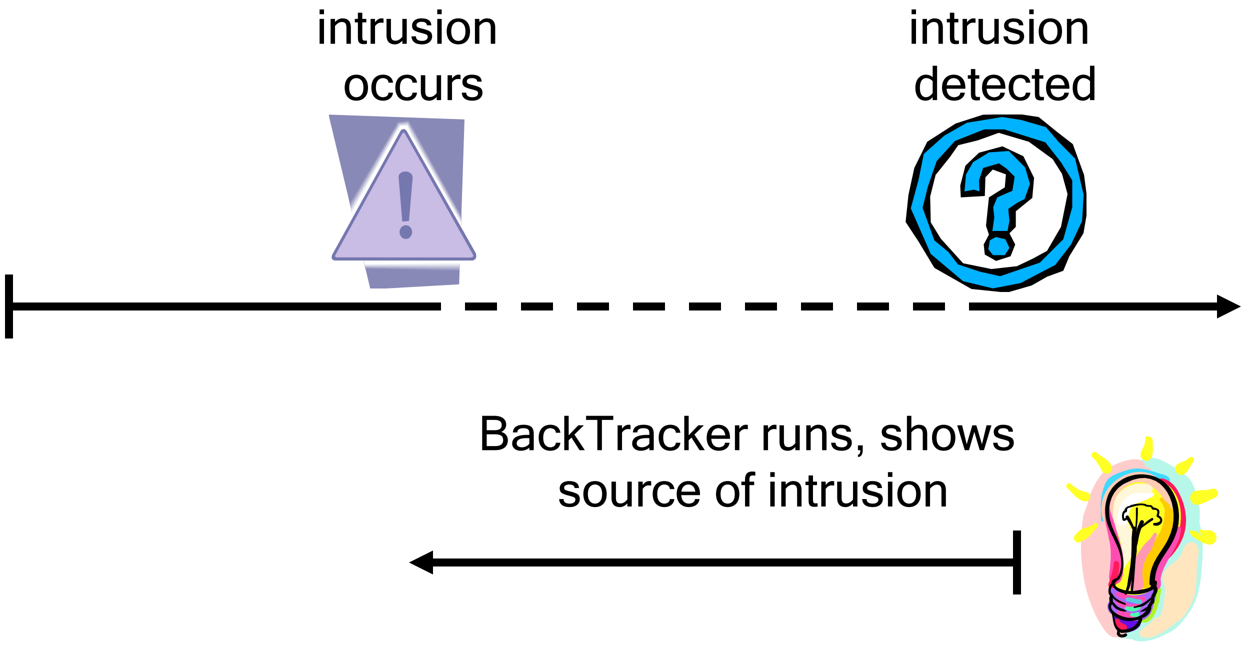


Backward Tracking
[King+, SOSP'03]

Step 3: Forward Tracking



BackTracker



- Online component logs objects and events
- Offline component generates graphs

BackTracker Objects

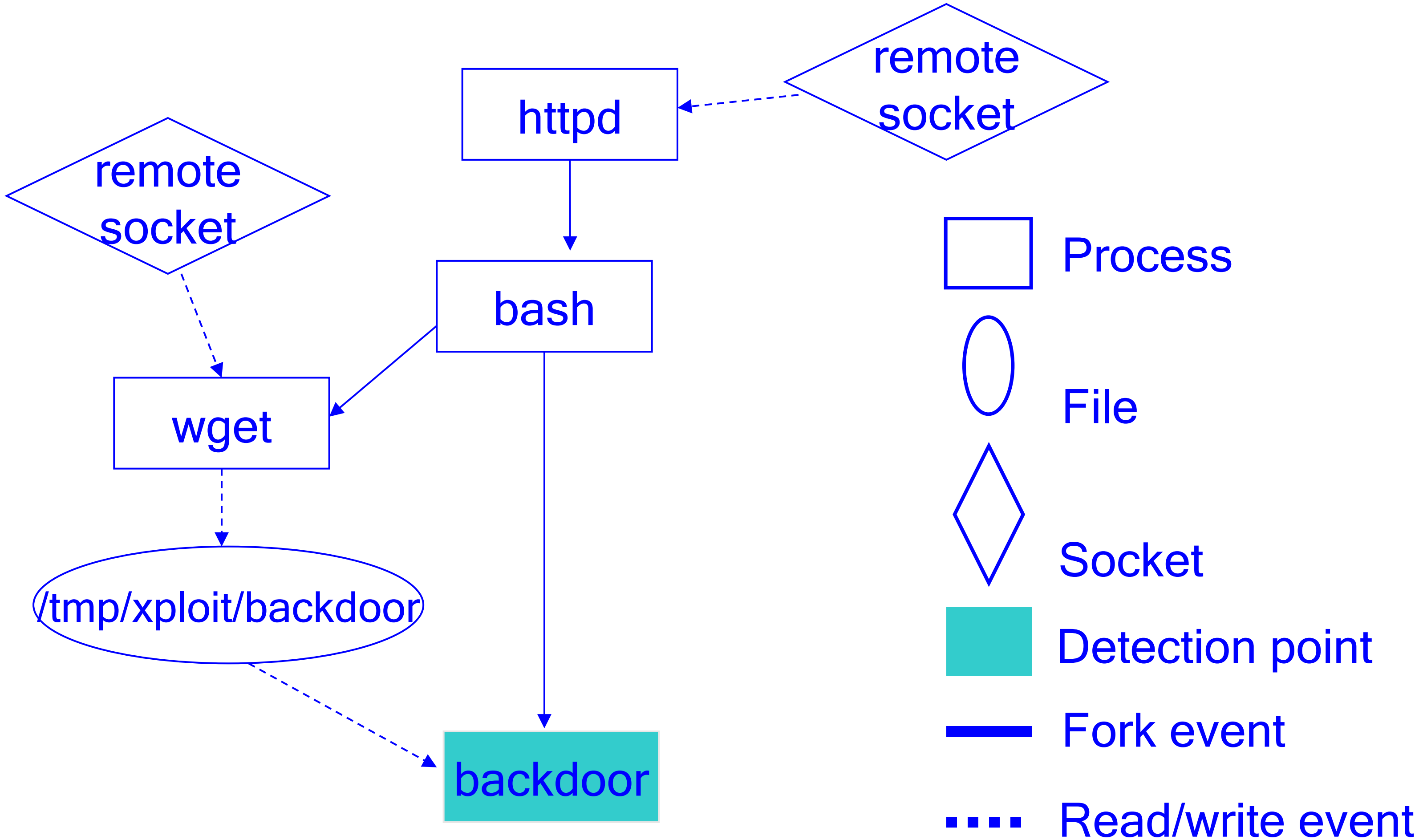
- Process
- File
- Filename

Dependency-Forming Events

- Process / Process
  - fork, clone, vfork
- Process / File
  - read, write, mmap, exec
- Process / Filename
  - open, creat, link, unlink, mkdir, rmdir, stat, chmod, …

- Dependency-tracking is an effective technique for highlighting actions of attacker

BackTracker Example

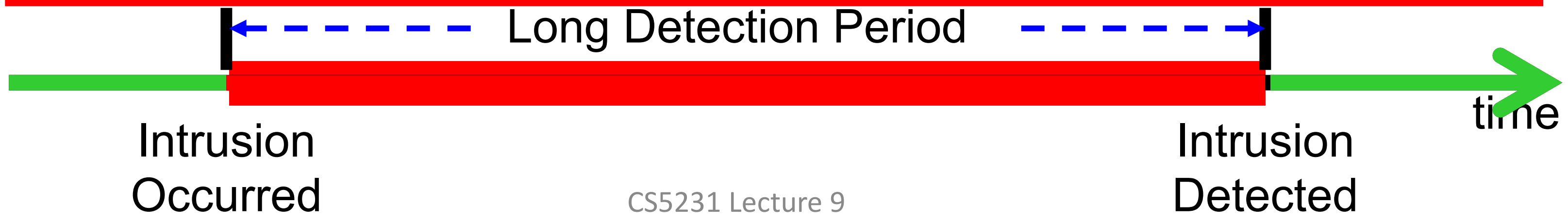


Challenge in Scalability

- Backward Tracking ➔ Break-in Point
  - Inputs: Detection Point, the Whole Log
- Forward Tracking ➔ Contaminations
  - Inputs: Break-in Point, the Whole Log

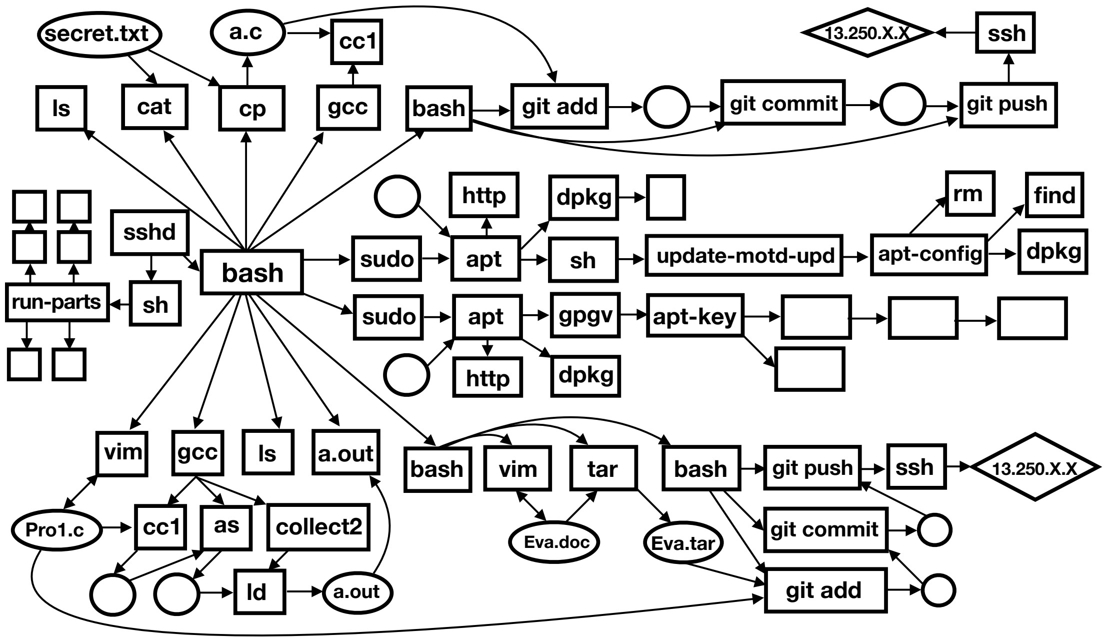
Analyze the whole log !

High Volume Log Data: 1.2 gigabytes per day for an intensive workload



Analysis of Provenance Graph

- Dependency analysis
- Subgraph matching
- Deep learning and recommendation



Related Work

- Scale up provenance analysis:
  - Data reduction [NDSS'16, 18 …] & Query system [Security'18, ATC'18 …]
  - Recognizing behaviors of interest requires intensive manual efforts

A semantic gap between low-level events and high-level behaviors

- Apply expert-defined specifications to bridge the gap
  - Match audit events against domain rules that describe behaviors
  - Query graph [VLDB'15, CCS'19], Tactics Techniques Procedures (TTPs) specification [SP'19,20], and Tag policy [Security'17,18]

Behavior-specific rules heavily rely on domain knowledge (**time-consuming)**

Related Work

- Scale up provenance analysis:
  - Data reduction [NDSS'16, 18 ...] & Query system [Security'18, ATC'18 ...]

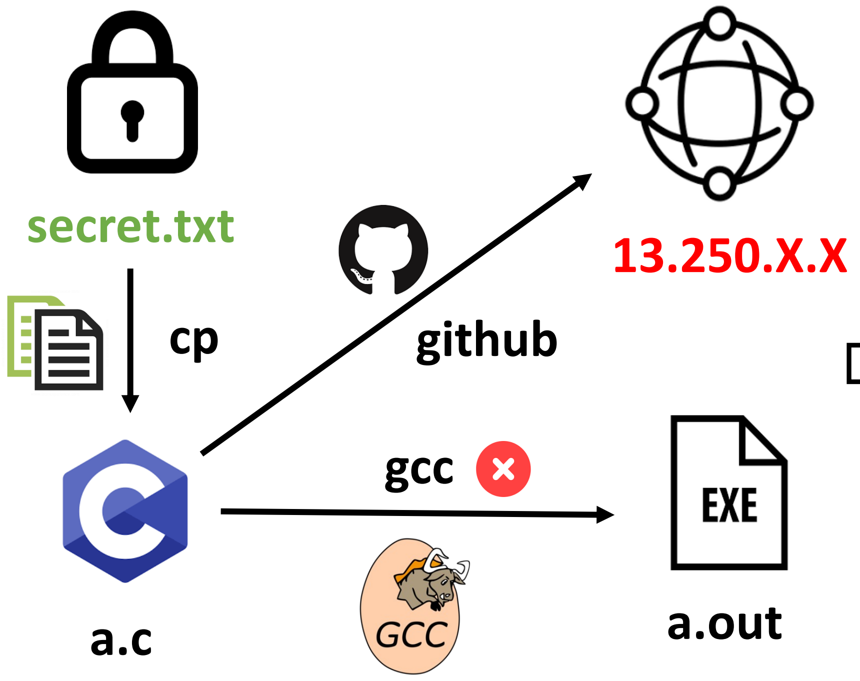
Can we automatically **abstract** high-level behaviors from low-level audit logs and **cluster** semantically similar behaviors before human inspection?

  - Query graph [VLDB'15, CCS'19], Tactics Techniques Procedures (TTPs) specification [SP'19,20], and Tag policy [Security'17,18]

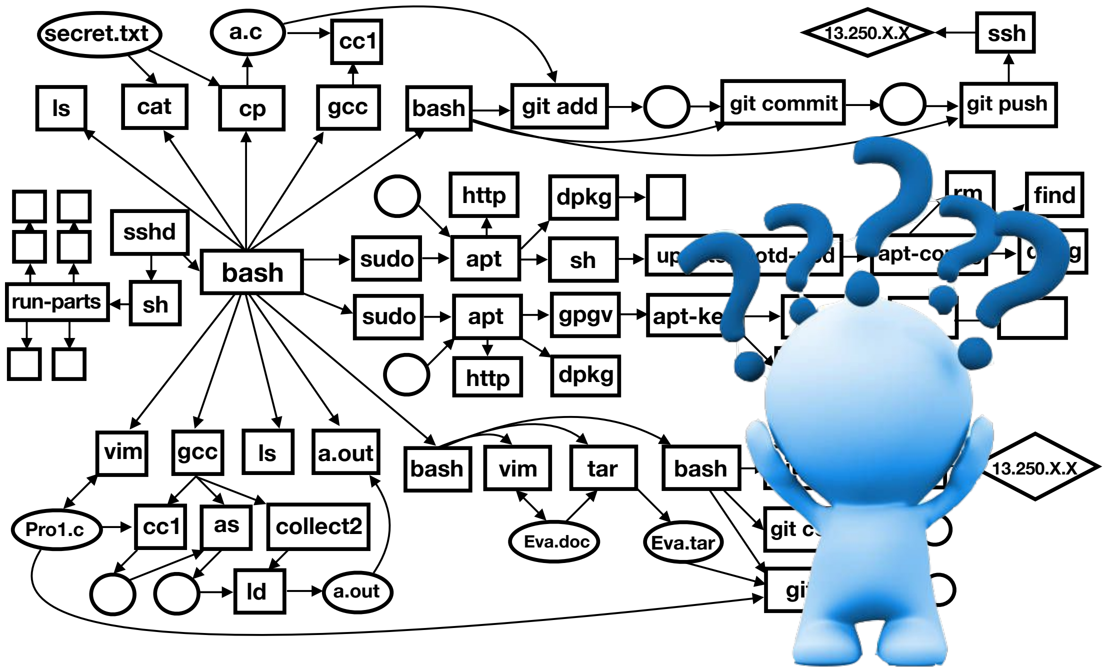
Behavior-specific rules heavily rely on domain knowledge (**time-consuming**)

Motivating Example

Attack Scenario: A software tester exfiltrates sensitive data that he has access to



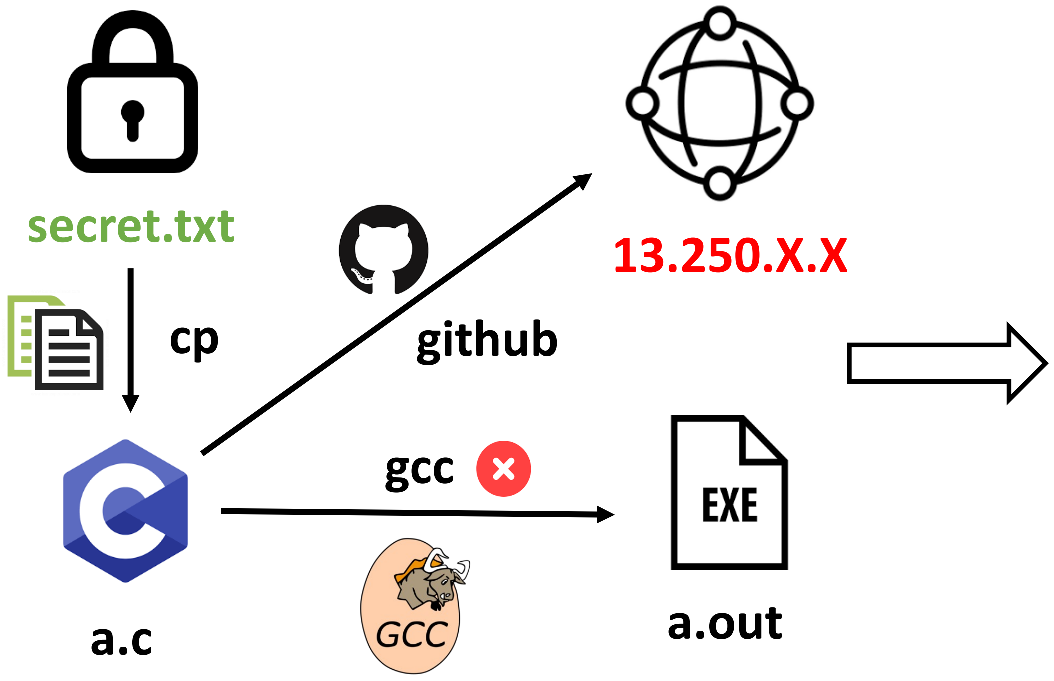
Data Exfiltration Steps



Motivating Example Logs

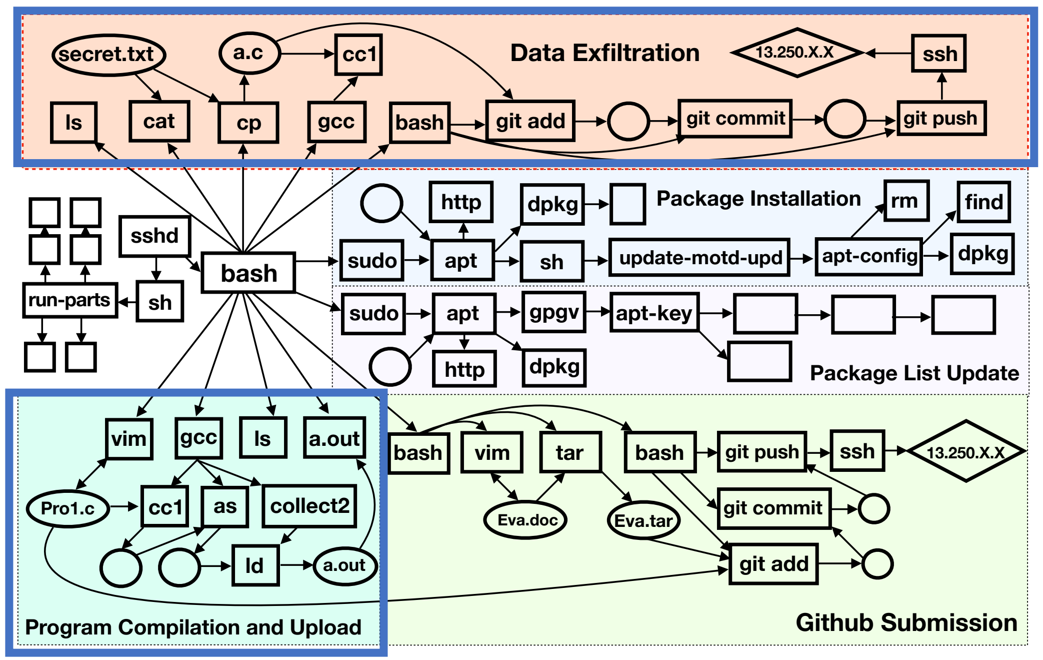
Motivating Example

Attack Scenario: A software tester exfiltrates sensitive data that he has access to



Data Exfiltration Steps

Data Exfiltration



Program Compiling and Upload (cluster)

Motivating Example Logs

Challenges for Behavior Abstraction

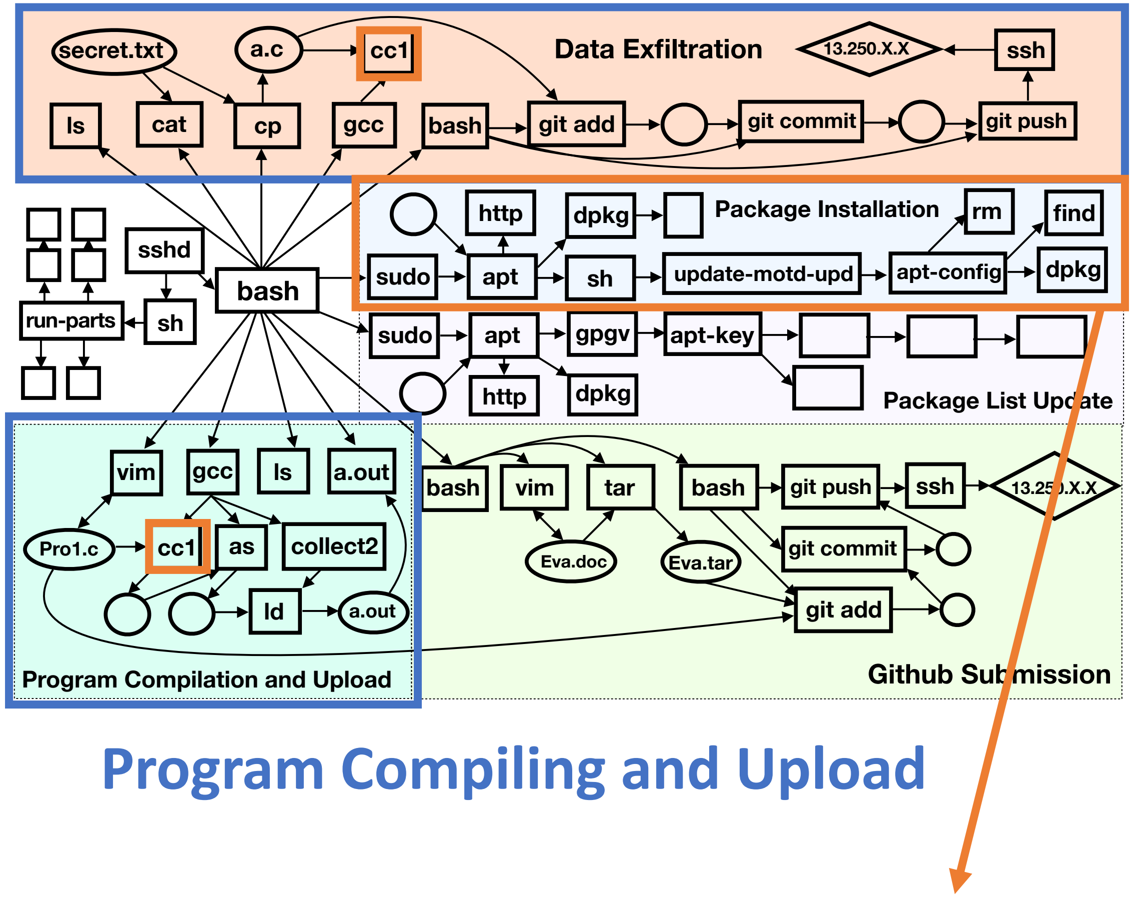
Event Semantics Inference:

- Logs record **general-purpose** system activities but lack knowledge of **high-level semantics**

Individual Behavior Identification:

- The volume of audit logs is **overwhelming**
- Audit events are **highly interleaving**

Data Exfiltration

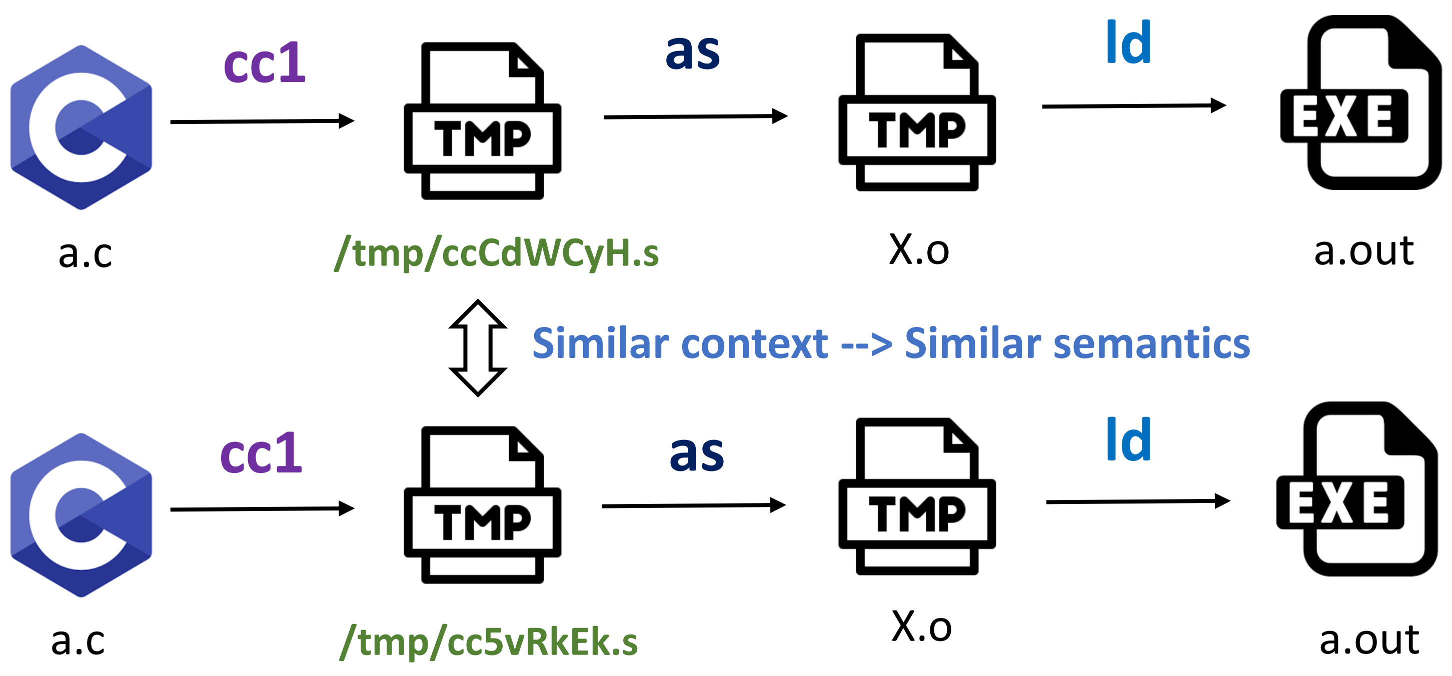


Program Compiling and Upload

Package Installation Events > 50,000

Our Insights

How do analysts manually interpret the semantics of audit events?

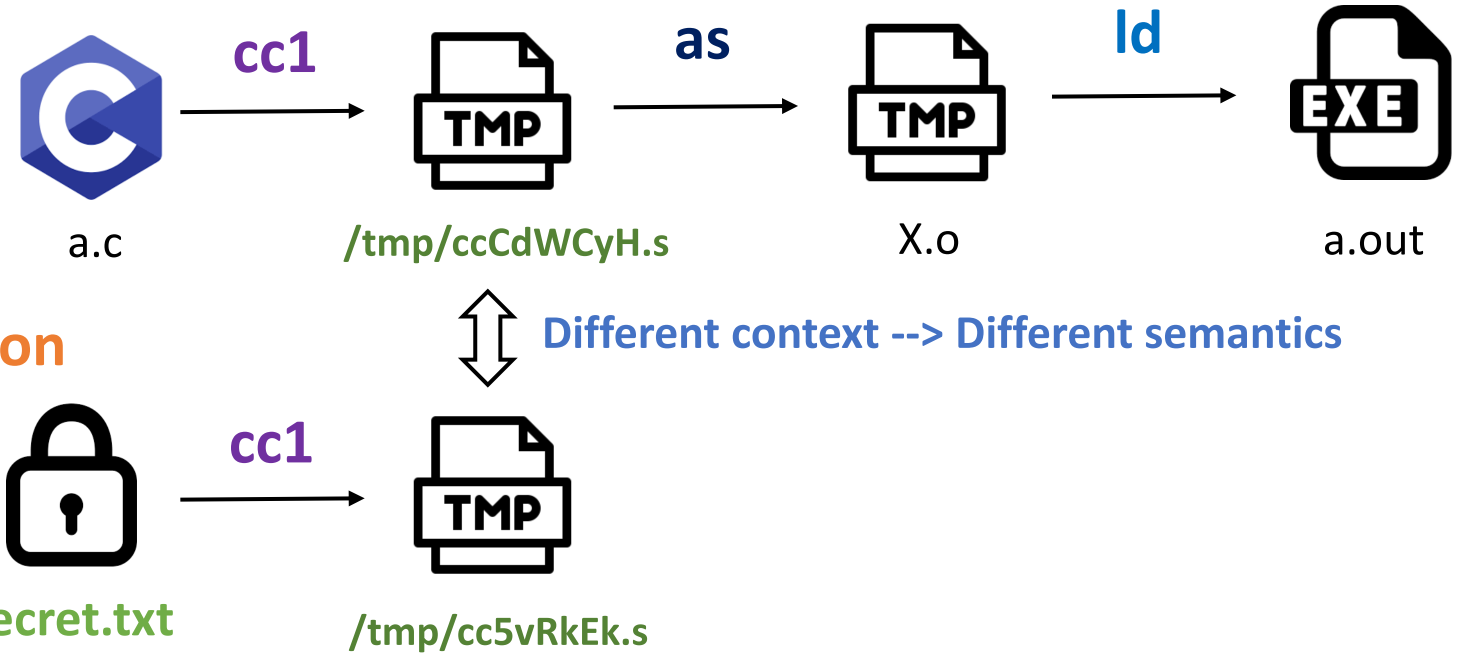


Compiling program using GCC

Our Insights

How do analysts manually interpret the semantics of audit events?

Data Exfiltration

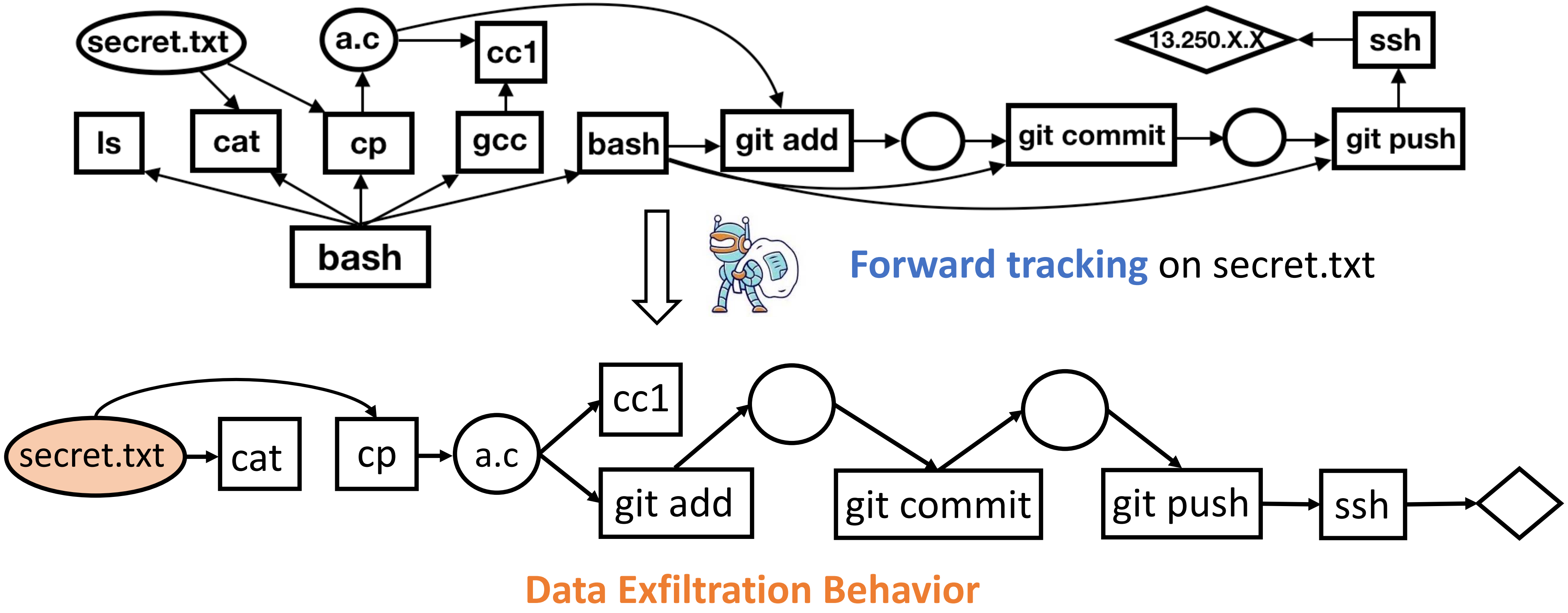


Compiling program using GCC

Reveal the semantics of audit events from their usage **contexts** in logs

Our Insights

How do analysts manually identify behaviors from audit events?

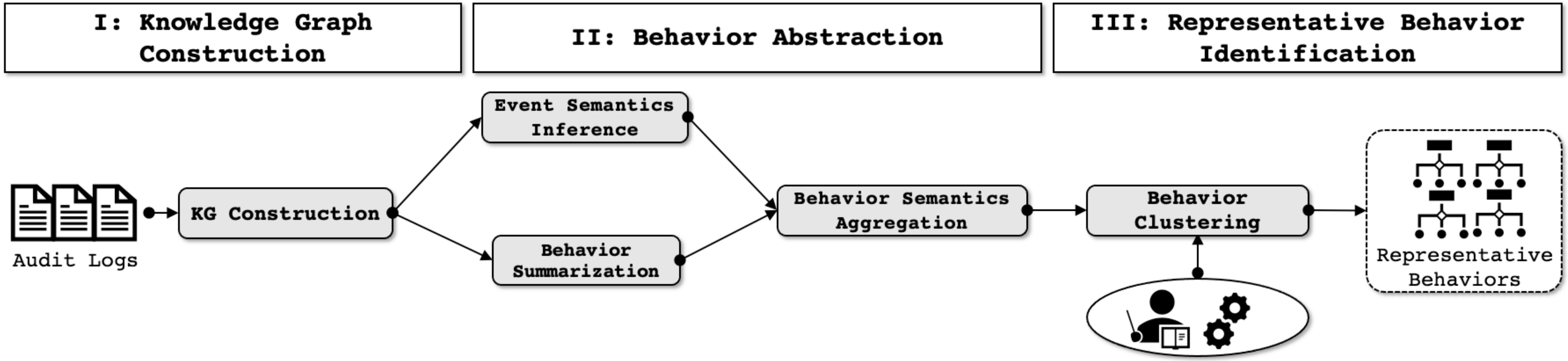


Summarize behaviors by tracking **information flows** rooted at **data objects**

WATSON

An automated behavior abstraction approach that **aggregates the semantics of audit logs to model behavioral patterns**

- Input: audit logs (e.g., Linux Audit[1])
- Output: representative behaviors



[1] Linux Kernel Audit Subsystem. https://github.com/linux-audit/ audit-kernel.

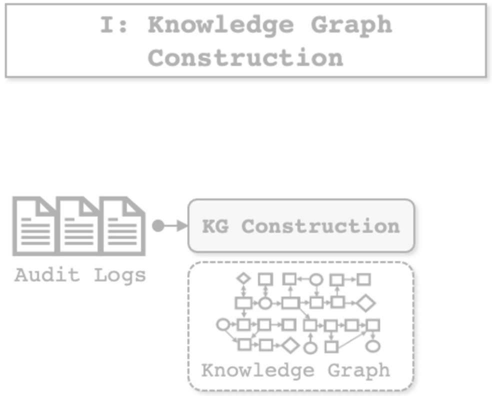
Knowledge Graph Construction

We propose to use a **knowledge graph** (KG) to represent audit logs:

- KG is a directed acyclic graph built upon triples
- Each triple, corresponding to an audit event, consists of three elements (head, relation, and tail):

$$\mathcal{KG} = \{(h, r, t) | h, t \in \{Process, File, Socket\}, r \in \{Syscall\}\}$$

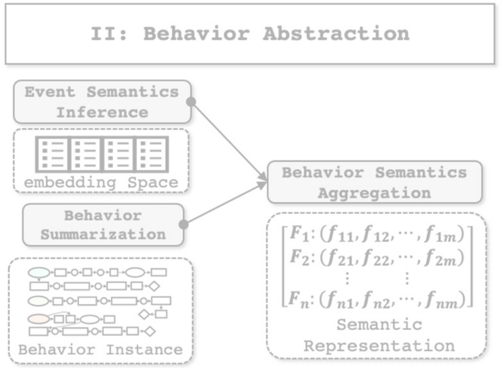
- KG unifies **heterogeneous** events in a **homogeneous** manner



Event Semantics Inference

- Suitable **granularity** to capture contextual semantics
  - Prior work [CCS'17] studies log semantics using events as basic units.
  - Lose contextual information within events
  - Working on **Elements** (head, relation, and tail) preserves more contexts

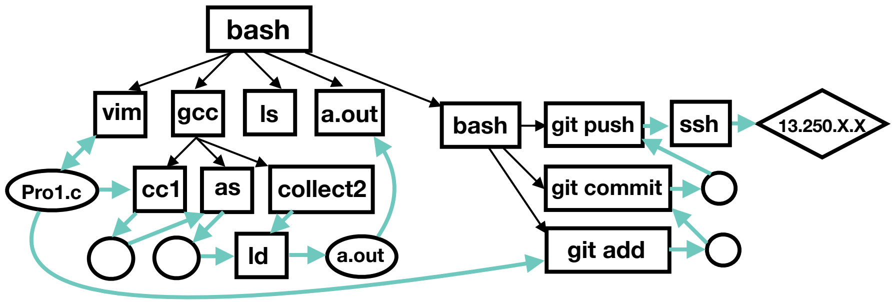
- Employ an embedding model to extract contexts
  - Map elements into a vector space
  - Spatial distance represents semantic similarities
  - **TransE:** a translation-based embedding model
  - **Head + Relation ≈ Tail → Context decides semantics**

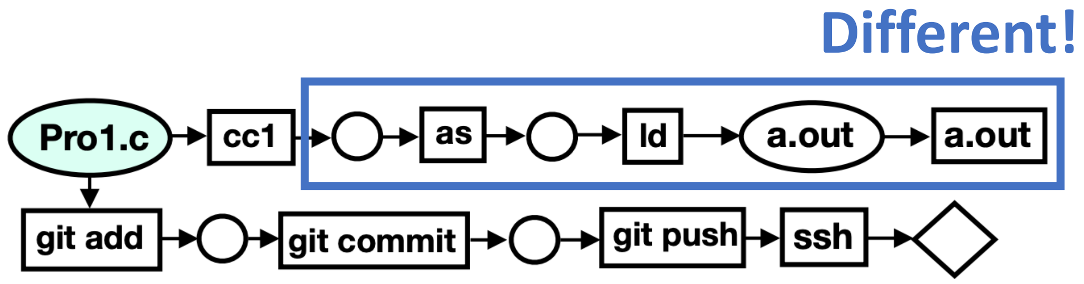


Behavior Summarization

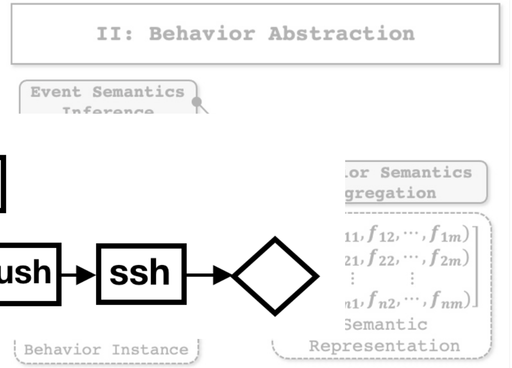
Individual behavior identification: Apply an adapted depth-first search (DFS) to track information flows rooted at a data object:

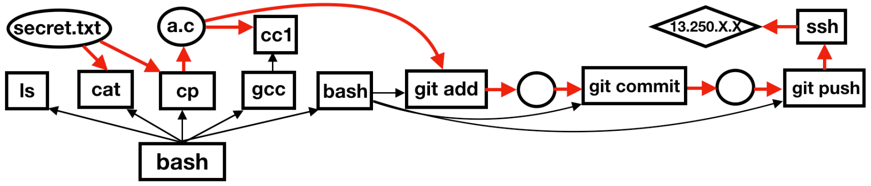
- Perform the DFS on every data object except libraries
- Two behaviors are merged if one is the subset of another

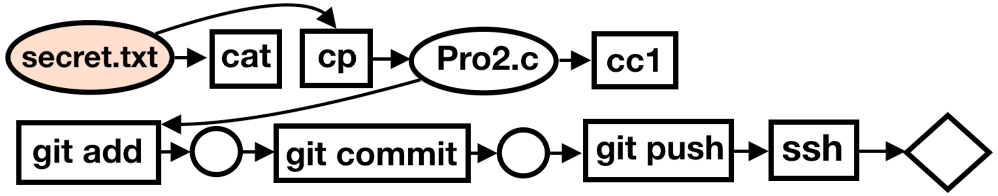




Program Compiling and Upload



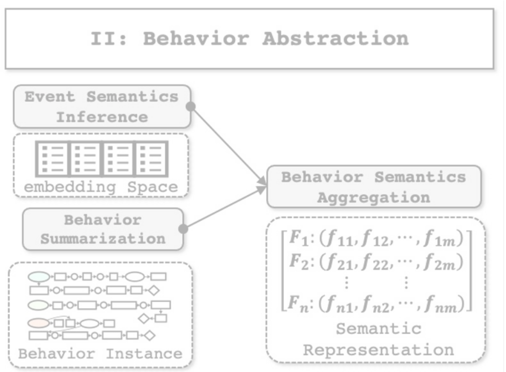




Data Exfiltration

Behavior Semantics Aggregation

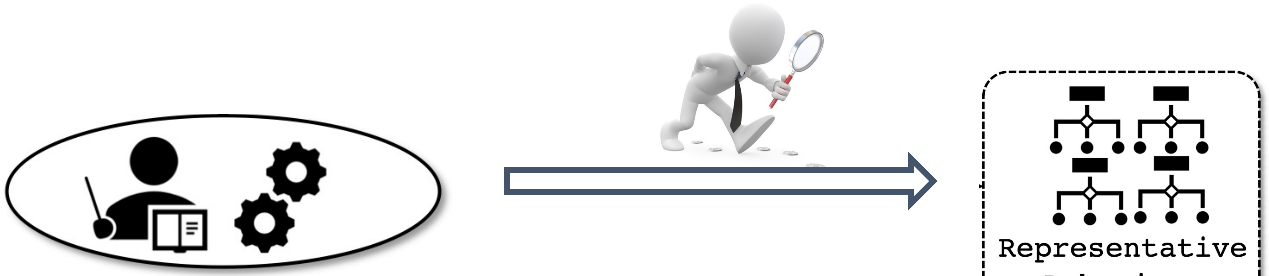
- How to aggregate event semantics to represent behavior semantics?
  - Naïve approach: Add up the semantics of a behavior’s constituent events
  - Assumption: audit events equally contribute to behavior semantics 
- Relative event importance
  - Observation: behavior-related events are common across behaviors, while behavior-unrelated events the opposite
  - Apply frequency as a metric to define event importance
  - Quantify the frequency: **Inverse Document Frequency (IDF)**
- The presence of **noisy events**
  - Redundant events [CCS’16] & Mundane events

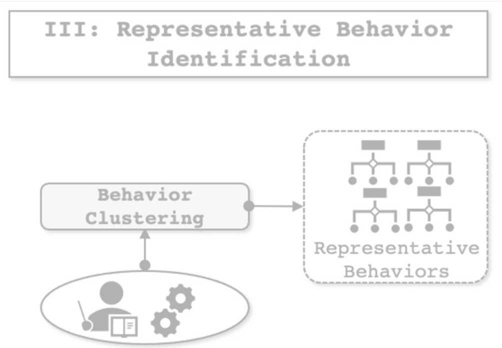


Representative Behavior Identification

- Cluster semantically similar behaviors: **Agglomerative Hierarchical Clustering analysis (HCA)**

- Extract the most representative behaviors
  - Representativeness: Behavior's average similarity with other behaviors in a cluster
  - **Analysis workload reduction**: Do not go through the whole behavior space





Summary

- Logging mechanisms
  - Application-level: Library wrapping / API hooking
  - Kernel-level: Syslogd/klogd, System call interception, Linux security module
  - Virtual Machine Monitor-level: System call interception
- Applications for auditing
  - Intrusion detection, recovery and investigation