Efficient Large-Scale Data Provenance Tracking and Analyzing: Intrusion Detection

Thomas Pasquier, University of Bristol
Two Sigma, 26/01/2021
Motivation: System call based intrusion detection

System Calls

- 
- 
- 
- 
- 
- 
- 
- 
- 
- 
Motivation: System call based intrusion detection

Identify abnormal patterns
Motivation: System call based intrusion detection

System Calls

1. Identify abnormal patterns
2. Hidden among benign actions
Motivation: System call based intrusion detection

System Calls

Identify abnormal patterns

Hidden among benign actions
Masquerading as benign action
Motivation: System call based intrusion detection

- Identify abnormal patterns
- Hidden among benign actions
- Masquerading as benign action
- Over a long period of time
What is provenance?
What is provenance?

- From the French “provenir” meaning “coming from”
- **Formal set of documents** describing the origin of an art piece
- **Sequence** of
  - Formal ownership
  - Custody
  - Places of storage
- Used for authentication
What is data-provenance?

- Represent interactions between objects of different types
  - Data-items (entities)
  - Processing (activities)
  - Individuals and Organisations (agents)
- Represented as a directed acyclic graph (think information flows)
- Edges represent interactions between objects as dependencies
- It is a representation of history
  - Immutable (unless it’s 1984)
  - No dependency to the future
Example provenance (simplified)
Example provenance (simplified)
Example provenance (simplified)

- **P1** (create)
- **S1**
- **P2** (read)
- **F1**

Connections:
- P1 to S1: create
- P2 to F1: read
Example provenance (simplified)
Example provenance (simplified)
Example provenance (simplified)
Example provenance (simplified)

Linux kernel compilation:
~2M graph elements
How is this useful?
Provenance-based security

● Provenance-based access control
  ○ A provenance-based access control model, IEEE PST 2012

● Loss Prevention Scheme
  ○ *Trustworthy Whole-System Provenance for the Linux Kernel, USENIX Security 2015

● Intrusion Detection
  ○ FRAPpuccino: fault-detection through runtime analysis of provenance, USENIX HotCloud 2017

● Moving towards complex runtime graph analysis
Provenance-based security

● Provenance-based access control
  ○ A provenance-based access control model, IEEE PST 2012

● Loss Prevention Scheme
  ○ *Trustworthy Whole-System Provenance for the Linux Kernel, USENIX Security 2015

● Intrusion Detection
  ○ FRAPpuccino: fault-detection through runtime analysis of provenance, USENIX HotCloud 2017

● Moving towards complex runtime graph analysis

● *overhead is a function of total graph size, a graph which grows indefinitely
  ○ 21ms overhead per network packet, on small graphs
Provenance-based intrusion detection

- **Intuition**: provenance graph exposes causality relationships between events
Provenance-based intrusion detection

- **Intuition:** provenance graph *exposes causality relationships* between events
Provenance-based intrusion detection

- Related events are connected even across long period of time
Concrete example: CI pipeline compromise

Han et al. “UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats”, NDSS 2020

- Attacker can control redirection when downloading through vulnerability
  - dependency packages.
- Install version of a tool used in the CI that contains a malware
- Modify the binary being generated during the CI compilation
- Binary is packaged, signed and distributed through legitimate channel

Difficulty:
- Each steps have very little abnormality (very close to normal behaviour)
- Causality is easily lost in complex build process

We continued work (with colleagues at NEC Labs) on malicious, but legitimate installer/package in:
How do we get the data?
Capture methods

Examples

3. Pasquier et al. "Practical whole-system provenance capture" SoCC. 2017
Capture methods

Examples

3. Pasquier et al. "Practical whole-system provenance capture" SoCC. 2017
Interposition is unsafe

- Watson "Exploiting Concurrency Vulnerabilities in System Call Wrappers" WOOT. 2007

- Time-of-audit-to-time-of-use attack
  - Race condition

- Syntactic Race
  - different copy of parameters

- Semantic Race
  - Kernel state may change
Capture methods

Examples

1. Based on Linux reference monitor
2. Best accuracy
3. Stronger formal guarantees
4. Formally specified semantic
5. Best performance

Pasquier et al. “Runtime Analysis of Whole-System Provenance”, CCS 2018
How do we process the data?
The problem

- We are building extremely large streaming graphs.
- As said earlier, previous solutions detection = f(size) …
- … won’t work in a runtime/streaming setting
The problem

- We are building extremely large streaming graphs.
- As said earlier, previous solutions detection = f(size) …
- … won’t work in a runtime/streaming setting

Pasquier et al. “Runtime Analysis of Whole-System Provenance”, CCS 2018
The solution

- Understand the properties of the graph (directed acyclic)
- Understand the semantic of the graph (OS execution)
- Understand the properties of the computation
  - Most can be translated as value propagation (e.g. to build feature vectors based on neighborhood)

Concretely in the implementation:

- Provide order guarantee
  - e.g. all incoming edge before outgoing, partial orders along paths etc.
  - Help with processing and garbage collection
- Use semantic for garbage collection
  - It is clear when nodes won’t be referenced again (e.g. inodes after free)
- Framework to write “query” based on value propagation
- In-kernel or userspace (same code)
  - Low level language, DSL would probably be better
How do we check we’ve done this properly?

- Static analysis of kernel + provenance capture instrumentation
  - Verify system calls semantic (manual)
  - Verify ordering

Figure 5: A whole-system provenance subgraph representing a valid instance of the model shown in Fig. 4.

Figure 4: Provenance model for the inode_post_setxattr hook.
How to perform detection?
Assumptions (and limitations)

- **Runtime detection**
  - We target environment with minimal human intervention
    - relatively consistent behaviour
    - e.g. web servers, CI pipelines etc...
  - Build a **model of system behaviour** (unsupervised training)
    - in a controlled environment
    - from a representative workload (this is hard!)
- **Detect deviation** from the model
- Several approaches being explored…
Example: UNICORN

- Han et al. “UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats”, NDSS 2020
Example: UNICORN

1) Graph streamed in, converted to histogram, labelled using (modified) **struct2vec**
Example: UNICORN

2) At regular interval, histogram converted to a fixed size vector using similarity preserving graph sketching
Example: UNICORN

3) Feature vectors are **clustered**
Example: UNICORN

4) Cluster forms “meta-state”, transitions are modelled
In deployment, anomaly detected via clustering and “meta-state” model
Relatively simple

- Labelled directed acyclic graph
  - node/edge types
  - security context (when available)
- Modification and combination of existing algorithms
  - struct2vec
  - similarity preserving hashing
  - clustering
- Right combination + domain knowledge
How to evaluate?
Comparison state of the art

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset</th>
<th># of Graphs</th>
<th>Avg.</th>
<th>Avg.</th>
<th>Preprocessed Data Size (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamSpot</td>
<td>YouTube</td>
<td>100</td>
<td>8,292</td>
<td>113,239</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>Gmail</td>
<td>100</td>
<td>6,827</td>
<td>37,382</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>Download</td>
<td>100</td>
<td>8,831</td>
<td>310,814</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>VGame</td>
<td>100</td>
<td>8,637</td>
<td>112,958</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>CNN</td>
<td>100</td>
<td>8,990</td>
<td>294,903</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>Attack</td>
<td>100</td>
<td>8,891</td>
<td>28,423</td>
<td>0.1</td>
</tr>
</tbody>
</table>

TABLE I: Characteristics of the StreamSpot dataset. The dataset is publicly available only in a preprocessed format.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamSpot (baseline)</td>
<td>0.74</td>
<td>N/A</td>
<td>0.66</td>
<td>N/A</td>
</tr>
<tr>
<td>$R = 1$</td>
<td>0.51</td>
<td>1.0</td>
<td>0.60</td>
<td>0.68</td>
</tr>
<tr>
<td>$R = 3$</td>
<td>0.98</td>
<td>0.93</td>
<td>0.96</td>
<td>0.94</td>
</tr>
</tbody>
</table>

TABLE II: Comparison to StreamSpot on the StreamSpot dataset. We estimate StreamSpot’s average accuracy and precision from the figure included in the paper [83], which does not report exact values. They did not report recall or F-score.


$R$ -> neighborhood size for struct2vec algorithm
Evaluation with DARPA datasets

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset</th>
<th># of Graphs</th>
<th>Avg.</th>
<th>Avg.</th>
<th>Raw Data Size (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>66</td>
<td>59,983</td>
<td>4,811,836</td>
<td>271</td>
</tr>
<tr>
<td>CADETS</td>
<td>Attack</td>
<td>8</td>
<td>386,548</td>
<td>5,160,963</td>
<td>38</td>
</tr>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>43</td>
<td>2,309</td>
<td>4,199,309</td>
<td>441</td>
</tr>
<tr>
<td>ClearScope</td>
<td>Attack</td>
<td>51</td>
<td>11,769</td>
<td>4,273,003</td>
<td>432</td>
</tr>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>2</td>
<td>19,461</td>
<td>1,913,202</td>
<td>4</td>
</tr>
<tr>
<td>THEIA</td>
<td>Attack</td>
<td>25</td>
<td>275,822</td>
<td>4,073,621</td>
<td>85</td>
</tr>
</tbody>
</table>

TABLE IV: Characteristics of graph datasets used in the DARPA experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DARPA CADETS</td>
<td>0.98</td>
<td>1.0</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>DARPA ClearScope</td>
<td>0.98</td>
<td>1.0</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>DARPA THEIA</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

TABLE V: Experimental results of the DARPA datasets.
Evaluation with DARPA datasets

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset</th>
<th># of Graphs</th>
<th>Avg.</th>
<th>Avg.</th>
<th>Raw Data Size (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>66</td>
<td>59,983</td>
<td>4,811,836</td>
<td>271</td>
</tr>
<tr>
<td>CADETS</td>
<td>Attack</td>
<td>8</td>
<td>386,548</td>
<td>5,160,963</td>
<td>38</td>
</tr>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>43</td>
<td>2,309</td>
<td>4,199,309</td>
<td>441</td>
</tr>
<tr>
<td>ClearScope</td>
<td>Attack</td>
<td>51</td>
<td>11,769</td>
<td>4,273,003</td>
<td>432</td>
</tr>
<tr>
<td>DARPA</td>
<td>Benign</td>
<td>2</td>
<td>19,461</td>
<td>1,913,202</td>
<td>4</td>
</tr>
<tr>
<td>THEIA</td>
<td>Attack</td>
<td>25</td>
<td>275,822</td>
<td>4,073,621</td>
<td>85</td>
</tr>
</tbody>
</table>

TABLE IV: Characteristics of graph datasets used in the DARPA experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>DARPA CADETS</td>
<td>0.98</td>
<td>1.0</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>DARPA ClearScope</td>
<td>0.98</td>
<td>1.0</td>
<td>0.98</td>
<td>0.99</td>
</tr>
<tr>
<td>DARPA THEIA</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

TABLE V: Experimental results of the DARPA datasets.

SUCH GOOD RESULTS ARE NOT NORMAL
Building our own dataset

- Attack designed to look similar to background activity
- Is that enough?

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset</th>
<th># of Graphs</th>
<th>Avg.</th>
<th>Avg.</th>
<th>Raw Data Size (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC-1</td>
<td>Benign</td>
<td>125</td>
<td>265,424</td>
<td>975,226</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>Attack</td>
<td>25</td>
<td>257,156</td>
<td>957,968</td>
<td>12</td>
</tr>
<tr>
<td>SC-2</td>
<td>Benign</td>
<td>125</td>
<td>238,338</td>
<td>911,153</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>Attack</td>
<td>25</td>
<td>243,658</td>
<td>949,887</td>
<td>12</td>
</tr>
</tbody>
</table>

TABLE VI: Characteristics of the datasets used in the supply-chain APT attack experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC-1</td>
<td>0.85</td>
<td>0.96</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>SC-2</td>
<td>0.75</td>
<td>0.80</td>
<td>0.77</td>
<td>0.78</td>
</tr>
</tbody>
</table>

TABLE VIII: Experimental results of the supply-chain APT attack scenarios.
Building our own dataset

- Attack designed to look similar to background activity

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset</th>
<th># of Graphs</th>
<th>Avg.</th>
<th>Avg.</th>
<th>Raw Data Size (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC-1</td>
<td>Benign</td>
<td>125</td>
<td>265,424</td>
<td>975,226</td>
<td>64</td>
</tr>
<tr>
<td>SC-1</td>
<td>Attack</td>
<td>25</td>
<td>257,156</td>
<td>957,968</td>
<td>12</td>
</tr>
<tr>
<td>SC-2</td>
<td>Benign</td>
<td>125</td>
<td>238,338</td>
<td>911,153</td>
<td>59</td>
</tr>
<tr>
<td>SC-2</td>
<td>Attack</td>
<td>25</td>
<td>243,658</td>
<td>949,887</td>
<td>12</td>
</tr>
</tbody>
</table>

TABLE VI: Characteristics of the datasets used in the supply-chain APT attack experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC-1</td>
<td>0.85</td>
<td>0.96</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>SC-2</td>
<td>0.75</td>
<td>0.80</td>
<td>0.77</td>
<td>0.78</td>
</tr>
</tbody>
</table>

TABLE VIII: Experimental results of the supply-chain APT attack scenarios.
Parameter Influence on detection performance

Figure 4: Detection performance (precision, recall, accuracy, and F-score) with varying hop counts (Fig. 4a), sketch sizes (Fig. 4b), intervals of sketch generation (Fig. 4c), and decay factor (Fig. 4d). Baseline values (*) are used by the controlled parameters (that remain constant) in each figure.
Processing Speed (overview)

Fig. 4: Total number of processed edges over time (in seconds) in the SC-1 experimental workload with varying batch sizes (Fig. 4(a)), sketch sizes (Fig. 4(b)), hop counts (Fig. 4(c)), and intervals of sketch generation (Fig. 4(d)). Dashed blue line represents the speed of graph edges streamed into UNASON for analysis. Triangle mean baseline has the same configurations as those used in our experiments and indicates the values of the controlled parameters (that remain constant) in each figure.

F. CPU & Memory Utilization

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Parameter Value</th>
<th>Max Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>H = 3</td>
<td></td>
<td>9457</td>
</tr>
</tbody>
</table>
Processing Speed (detail)
CPU and memory usage

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Parameter Value</th>
<th>Max Memory Usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hop Count</td>
<td>R = 1</td>
<td>562</td>
</tr>
<tr>
<td></td>
<td>R = 2</td>
<td>624</td>
</tr>
<tr>
<td></td>
<td>R = 3</td>
<td>687</td>
</tr>
<tr>
<td></td>
<td>R = 4</td>
<td>749</td>
</tr>
<tr>
<td></td>
<td>R = 5</td>
<td>812</td>
</tr>
<tr>
<td>Sketch Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sketch Size</td>
<td>S = 500</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>S = 1,000</td>
<td>437</td>
</tr>
<tr>
<td></td>
<td>S = 2,000</td>
<td>687</td>
</tr>
<tr>
<td></td>
<td>S = 5,000</td>
<td>1,374</td>
</tr>
<tr>
<td></td>
<td>S = 10,000</td>
<td>2,498</td>
</tr>
</tbody>
</table>

Table 5: Memory usage with varying hop counts and sketch sizes.

Figure 6: Per virtual CPU and average CPU utilization.
Long term CPU usage

CPU over long time period? 15% CPU time across cores

Figure 5: Average CPU utilization with the baseline configurations.
Some insights from this work
We can build practical provenance-based IDSs

- We can detect intrusion out of graph structure with little metadata
  - Vertex type (thread, file, socket etc…)
  - Edge type (read, write, connect etc…)
- Processing speed
  - Current prototype
  - Data generation speed < processing speed!
Proper evaluation is hard!

- Dataset are hard to generate
  - What is a good quality dataset?
- Hard to compare across papers, a lot is not available
  - Experiments (i.e. attacks)
  - Capture Mechanisms
  - Analysis pipelines
- Leads to unsatisfactory evaluation
  - I may be able to compare to similar techniques (may reuse dataset)
  - ... very hard for unrelated one (i.e. ingest different data type)
- Adversarial ML?
Identifying threats: explainability is a problem

- There is a problem within the last batch of X graph elements
  - 2,000 in previous figures
- Good luck finding out what went wrong
- Provenance forensic is an active field of research
  - Promising work out of the DARPA programme
- … but could we do better during detection?
Other approaches?
Does my system do what I think it should?

Pasquier et al. “Data provenance to audit compliance with privacy policy in the Internet of Things”, Personal and Ubiquitous Computing, 2017
Some move in that direction (sort-ish)

Can we get there?
Thank you! Questions?

tfjmp.org
CamFlow capture mechanism

- Leverage existing kernel features whenever possible
- Avoid alteration of existing code
- We therefore build upon:
  - **Linux Security Module**
    - to capture system events
  - **NetFilter**
    - to capture network events
  - **RelayFS**
    - to transfer provenance to user space
  - **SecurityFS**
    - to provide a userspace interface for settings
## Extent of modification

### Modifications to the Linux Kernel code

<table>
<thead>
<tr>
<th>System</th>
<th>Headers</th>
<th>C File</th>
<th>Total</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS (v2.6.27) pub. 2006</td>
<td>18</td>
<td>69</td>
<td>87</td>
<td>5100</td>
</tr>
<tr>
<td>LPM (v2.6.32) pub. 2015</td>
<td>13</td>
<td>61</td>
<td>74</td>
<td>2294</td>
</tr>
<tr>
<td>CamFlow (v5.4.15) circa 2020</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>4220</td>
</tr>
</tbody>
</table>
## Capture overhead

### Micro-benchmark

<table>
<thead>
<tr>
<th>Sys Call</th>
<th>Whole</th>
<th>Selective</th>
</tr>
</thead>
<tbody>
<tr>
<td>stat</td>
<td>100%</td>
<td>28%</td>
</tr>
<tr>
<td>open/close</td>
<td>80%</td>
<td>18%</td>
</tr>
<tr>
<td>fork</td>
<td>6%</td>
<td>2%</td>
</tr>
<tr>
<td>exec</td>
<td>3%</td>
<td>&lt;1%</td>
</tr>
</tbody>
</table>

**Selective:** cost of allocating/freeing provenance “blob” + recording or not decision

**Whole:** Selective + cost of recording provenance information

### Macro-benchmark

<table>
<thead>
<tr>
<th>Prog.</th>
<th>Whole</th>
<th>Selective</th>
</tr>
</thead>
<tbody>
<tr>
<td>unpack</td>
<td>2%</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>build</td>
<td>2%</td>
<td>0%</td>
</tr>
<tr>
<td>postmark</td>
<td>11%</td>
<td>6%</td>
</tr>
</tbody>
</table>

**Selectivity**

- **Micro-benchmark**
  - stat: 100% vs 28%
  - open/close: 80% vs 18%
  - fork: 6% vs 2%
  - exec: 3% vs <1%

- **Macro-benchmark**
  - unpack: 2% vs <1%
  - build: 2% vs 0%
  - postmark: 11% vs 6%
Advanced Persistent Threats

Reconnaissance
Identify Target & Explore Vulnerabilities

Weaponize
Design Backdoor & Penetration Plan

Delivery
Deliver the Weapon

Exploitation
Victim Triggers Vulnerability

Installation
Install Backdoor or Malware

Command & Control
Give Remote Instructions to Victim

Actions on Objectives

Zero-Day Exploits

Diverse Attack Vectors

- Active Scanning
- Passive Scanning
- Malware
- Scripting
- Spearphishing
- Supply-chain Attack
- Application Shimming
- Job Scheduling
- Hooking
- Dylib Hijacking
- Connection Proxy
- Domain Fronting

Long Duration

Low-and-Slow Attack Patterns