Automated In-memory Malware/Rootkit Detection via Binary Analysis and Machine Learning

By: Malachi Jones, PhD
### ABOUT ME

**Education**
- **Bachelors Degree**: Computer Engineering (Univ. of Florida, 2007)
- **Master’s Degree**: Computer Engineering (Georgia Tech, 2009)
- **PhD**: Computer Engineering (Georgia Tech, 2013)

**Cyber Security Experience**
- **Harris Corp.**: Cyber Software Engineer/ Vuln. Researcher (2013-2015)
- **Booz Allen Dark Labs**: Embedded Security Researcher (2016- Present)
# Appendix

- Machine Learning Primer
- Advanced Binary Analysis
- Memory Forensics
- Binary Vector Generation
- Call Trace Vector Generation
- Hooking
**Motivation**

- **Code injection** is a technique utilized by malware to *hide malicious code in a legitimate process and/or library* and to force a legitimate process to perform the execution on the malware’s behalf.

- In addition to **PE Injection** and **DLL Hijacking** *(shown above)*, other methods include **process hollowing** and **reflective DLL Injection**.
**Motivation**

- **Emulation** and **Hooking** are modern techniques that are employed by anti-virus (AV) vendors (shown above) to monitor the execution behavior of binaries executing on a target host.

- These techniques combined with **Execution Behavior Analysis** can allow for the discovery of **Advance Persistent Threats** (APT)s that leverage advanced **code injection techniques** to hide in memory and disguise execution.
Motivation

- **Problem:** Hooking and “traditional” emulation techniques can be reliably evaded by APTs

- **Examples:**
  
  - **Hooking** - Malware can either use lower level unhooked APIs or remove hooks at runtime
  
  - **Emulation** - Utilize an incorrectly implemented API emulation function (e.g. undocumented Windows API) and detect unexpected output given a specified input
**Observation**: A necessary condition for malicious code to be executed is that the code *must reside in memory prior to and during execution*

*As a consequence, periodic live collection and analysis of memory artifacts can provide an effective means to identify malware residing on a host*
**Objective**

- **Demonstrate** an approach to compare the following memory artifacts across a set of networked hosts *in a scalable manner* to identify anomalous code:
  i. Processes
  ii. Shared Libraries
  iii. Kernel Modules
  iv. Drivers

- Specifically, we will discuss how an approximate clustering algorithm with linear run-time performance can be leveraged to identify outliers among a set of equivalent types of artifacts (e.g. explorer.exe processes) collected from each networked host.

- *We will also discuss how Dynamic Binary Analysis can be utilized to improve detection of sophisticated malware threats.*
**Main Takeaways**

- **Phase I**: We’ll leverage **static analysis** to provide us with a computationally efficient way to rapidly identify memory artifacts with anomalous code.

- **Phase II**: Dynamic Analysis will be utilized to differentiate between **benign** anomalous code and **malware**.
**Main Takeaways**

**Set of Networked Hosts**

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

An agent can be installed on each host to periodically send memory artifacts to a collection server.

**Memory Artifacts Sent to Collection Server**

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

An example type of artifact is an explorer.exe process.

**Identical types of artifacts are clustered**

- Host D
- Host B
- Host G

- Host F

Outliers (e.g. Host F) are analyzed dynamically to more accurately identify malicious behavior.
Main Takeaways

Set of Networked Hosts

Host A  Host B  Host C  Host D  Host E  Host F  Host G

An agent can be installed on each host to periodically send memory artifacts to a collection server.

Memory Artifacts Sent to Collection Server

Host A  Host B  Host C  Host D  Host E  Host F  Host G

An example type of artifact is an explorer.exe process.

Identical types of artifacts are clustered

Host D  Host B  Host G  Host F  Host E  Host A  Host C

Outliers (e.g., Host F) are analyzed dynamically to more accurately identify malicious behavior.
**Main Takeaways**

**Set of Networked Hosts**

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

An agent can be installed on each host to periodically send memory artifacts to a collection server.

**Memory Artifacts Sent to Collection Server**

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

An example type of artifact is an explorer.exe process.

**Identical types of artifacts are clustered**

- Host D
- Host B
- Host G
- Host F
- Host E
- Host A
- Host C

Outliers (e.g. Host F) are analyzed dynamically to more accurately identify malicious behavior.
**Main Takeaways**

Set of Networked Hosts

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

*An agent can be installed on each host to periodically send memory artifacts to a collection server*

Memory Artifacts Sent to Collection Server

- Host A
- Host B
- Host C
- Host D
- Host E
- Host F
- Host G

*An example type of artifact is an explorer.exe process*

Identical types of artifacts are clustered

- Host D
- Host B
- Host G
- Host F
- Host E
- Host A
- Host C

*Outliers (e.g. Host F) are analyzed dynamically to more accurately identify malicious behavior*
Rekall is an open source tool that can be used to acquire live memory and perform analysis.

We can develop an agent that is deployed on each host that interfaces with Rekall to collect desired memory artifacts in an automated fashion.
- Querying for active and terminated processes

**pslist** command allows for live querying of internal data structures in memory for active and terminated processes.
Memory Acquisition Automation

- Querying for active and terminated processes

**pslist** command allows for live querying of internal data structures in memory for active and terminated processes.

Terminated process (notepad++) still in memory 26 days later.
### Memory Acquisition Automation

- Capturing live dumps of binaries w/ Rekall

<table>
<thead>
<tr>
<th>_EPROCESS</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfa800d9deb10 smss.exe</td>
<td>324 executable.smss.exe_324.exe</td>
</tr>
<tr>
<td>0xfa80106af8e0 notepad++.exe</td>
<td>332 executable.notepad.exe_332.exe</td>
</tr>
<tr>
<td>0xfa800ef96060 stacsv64.exe</td>
<td>368 executable.stacsv64.exe_368.exe</td>
</tr>
<tr>
<td>0xfa800d663460 AmazonDrive.exe</td>
<td>380 executable.AmazonDrive.exe_380.exe</td>
</tr>
</tbody>
</table>

**procdump** dumps a set of specified processes (given pids as input) to a desired directory
Capturing live dumps of binaries w/ Rekall

`procdump` dumps a set of specified processes (given pids as input) to a desired directory
DEMO MEMORY ACQUISITION AUTOMATION

Windows Client

Ubuntu Server
(DEMO) MEMORY ACQUISITION AUTOMATION

Process list queried and sent to server

Server processing client’s process list
(DEMO) MEMORY ACQUISITION AUTOMATION

Dumped processes received from client
(Demo) Memory Acquisition Automation

Process dumps disassembled with Binary Ninja
PHASE I DETECTION: STATIC ANALYSIS
## Phase I Detection: Static Analysis

### Overall Goals:

- **Group memory artifacts based on their similarity** (as demonstrated in the above figure) **to identify outliers**
- **Leverage Dynamic Analysis (Phase II) to differentiate between benign anomalous code and malware**

### Memory Artifacts Sent to Collection Server

<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
<th>Host C</th>
<th>Host D</th>
<th>Host E</th>
<th>Host F</th>
<th>Host G</th>
</tr>
</thead>
</table>

Identical types of artifacts are clustered:

<table>
<thead>
<tr>
<th>Host D</th>
<th>Host B</th>
<th>Host G</th>
<th>Host F</th>
<th>Host E</th>
<th>Host A</th>
<th>Host C</th>
</tr>
</thead>
</table>
Requirements for clustering artifacts at scale

- Computationally efficient method for determining the similarity (or dissimilarity) of a pair of binaries
- Clustering algorithm with a linear run-time performance in the worst-case
PHASE I DETECTION: STATIC ANALYSIS

Computationally Efficient Differing Algorithm
**Efficient Diffing Algorithm**

- **Question:** Why is efficient binary diffing *critical* to our goal of detecting malicious code?
  
  - Slow diffing ➔ Slow clustering ➔ Delayed threat detection
  
  - We want to be able to **cluster a large set of binaries (10,000+)** pretty quickly to identify binaries that pose potential threats to hosts on the network
  
  - Clustering algorithms need to **perform a large number of diffing** operations with respect to the binaries (exact number depends on run-time algorithm performance)
# Efficient Diffing Algorithm

## Step 1: Generate a vector representation of each binary

**main (explorer.exe)**

- `push ebp`
- `mov rbp, rsp`
- `sub rsp, 0x10`

Each row of the vector represents the number of occurrences of a unique sequence of instructions.

## Step 2: Compute the similarity of a pair of vectors

**Similarity Function**

- `Similarity Function` takes as input two vectors and produces a value between 0 and 1.

(See Appendix D for more details about the algorithm)
Efficient Diffing Algorithm

- Evaluating similarity function of on-disk copy of `explorer.exe` vs. in-memory image
Efficient Differing Algorithm

- Evaluating similarity function against explorer.exe

```
/home/targaryen/.virtualenvs/angr/bin/python /targaryen/targaryen/BinaryReportAnalyzer/ReportSimilarityAnalysis:
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:
 Report Similarity Analysis
 file 'executable.1844.exe'
 hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a6287044d17690a75bf628'
 file 'explorer_memory_forensics.exe'
 hash:'6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef9dab53aef1cf0f'
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Finished Jaccard index analysis (Elapsed time:0.0579369068146)
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jaccard index: **0.834103965252**

Process finished with exit code 0
```

Jaccard Index: 83.4% similar
**Efficient Diffing Algorithm**

- Comparing similarity results against BinDiff

**BinDiff: 84% similar**

---

**Diff Info**
- **Diff Path**: C:\Users\malachi\Documents\Presentations\My Presentations\2017\Targaryen\demo\Demo\Demo.BinDiffWorkspace - BinDiff
- **File Date**: Aug 15, 2017 9:07:04 PM

**Primary Image**
- **IDB Name**: explorer_memory_forensics (54Bit)
- **Image Name**: explorer_memory_forensics.exe
- **Hash**: 9BD0992115A03FCDDF3DFEA5E4859A646
- **Architecture**: x86-64
- **Functions**: 3172 (78.3%), 4051 (21.7%), 879

**Secondary Image**
- **IDB Name**: executable.1844 (54Bit)
- **Image Name**: executable.1844.exe
- **Hash**: 274CA2D50EFFF79CF55016F9696641F6557
- **Architecture**: x86-64
- **Functions**: 3172 (99.9%), 3174 (0.1%), 2
Efficient Diffing Algorithm

- Comparing similarity results against BinDiff

  hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a628704

INFO:BinaryReportAnalyzer.ReportAnalysis:
  (Elapsed Time: 0.0328039596243) Deserialized report analysis file 'explorer_memory_forensics.exe'
  hash:'6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef

INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Finished Jacard index analysis (Elapsed time:
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jacard index: 0.834103965252

BinDiff: 84% similar
Jaccard Index: 83.4% similar
Efficient Diffing Algorithm

- **Performance vs BinDiff**
  
  ```
  INFO:BinaryReportAnalyzer.ReportAnalysis:
  (Elapsed Time: 0.0248849391937) Deserialized report analysis
  file 'executable.1844.exe'
  hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a6287044d17690a75bf628'
  
  INFO:BinaryReportAnalyzer.ReportAnalysis:
  (Elapsed Time: 0.0328030586243) Deserialized report analysis
  file 'explorer_memory_forensics.exe'
  hash:'6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef9dab53aeff1c00'
  
  (Elapsed time:0.0579369068146)
  INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jaccard index: 0.834103965252
  
  Jaccard Performance: 57 ms
  ```

  - **BinDiff Analysis** : 4.2 seconds
  - **Jaccard Index Analysis**: 57 ms (in pure python)
  - **Speed up factor of 74**
Clustering Algorithms

- We’ll utilize an **agglomerative hierarchical clustering algorithm** because we don’t have to specify the exact number of clusters, $k$, a priori (vs. k-means, where $k$ must be specified).

- Computational complexity for a non-approximated implementation of the algorithm is $O(n^2 \log n)$, which is not a desirable property for achieving scalability.

- Instead, we’ll use an **approximate implementation** (presented in [1]), which has a **linear worse-case complexity** of $O(k^2 \log k + n)$.

- **Note**: $k$ is constant and $k << n$
Approximate Clustering Algorithm [1] Sketch

**Step 1: Prototype Extraction** \[O(k \cdot n)\]

Prototypes are a small \((k << n)\), yet representative subset of artifacts.

**Step 2: Clustering w/ Prototype** \[O(k^2 \log (k) + n)\]
**Cluster Artifacts At Scale**

- Prototype Extraction Algorithm [1]

```
Algorithm 1 Prototype extraction

1: prototypes ← ∅
2: distance[x] ← ∞ for all x ∈ reports
3: while max(distance) > d_p do
4:   choose z such that distance[z] = max(distance)
5: for x ∈ reports and x ≠ z do
6:   if distance[x] > ||φ(x) - φ(z)|| then
7:     distance[x] ← ||φ(x) - φ(z)||
8: add z to prototypes
```
PHASE II DETECTION: DYNAMIC ANALYSIS
Although dynamic analysis can be more accurate (i.e. fewer false-positives) than static analysis, it can also be more computationally expensive.

Therefore, we’ve discussed utilizing static analysis (Phase I) to filter out binaries based on their similarity.

Consequently, we can focus computationally efforts in Phase II on differentiating benign anomalous code from APTs via dynamic analysis.
**Phase II: Dynamic Analysis**

- **Key Concept:** Binaries utilize system calls and library function calls (e.g. dlls) to interact with the operating system in order to perform meaningful/desired operations

- **Corollary:** By analyzing the external call sequences of a binary (e.g. call trace), the underlying behavior and intent of the binary can be characterized
Case Study: “Sample J” Malware

(sha1: 70cb0b4b8e60dfed949a319a9375fac44168ccbb)
Case Study: “Sample J” Malware

```c
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, 
{
    HANDLE v4;    // edi@4
    DWORD v5;    // eax@10
    DWORD v6;    // ecx@10
    PROCESSENTRY32 pe;    // [esp+4h] [ebp-130h]@4
    char v8[6];    // [esp+12Ch] [ebp-8h]@1

    _sidt(v8);
    if ( *(DWORD *)v8[2] > 0x8000F400 && *(DWORD *)v8[2] < return 0;
    pe.dwSize = 0;
    memset(&pe.cntUsage, 0, 0x124u);
    v4 = CreateToolhelp32Snapshot(2u, 0);
    if ( v4 == (HANDLE)-1 )
        return 0;
    pe.dwSize = 296;
    if ( Process32First(v4, &pe) )
    {  if (!stricmp(pe.szExeFile, Str2) )
      {
        LABEL_10:
          v5 = pe.th32ParentProcessID;
          v6 = pe.th32ProcessId;
          goto LABEL_12;
        }
        while ( Process32Next(v4, &pe) )
        {  if (!stricmp(pe.szExeFile, Str2) )
            goto LABEL_10;
          }
      }
    v5 = fdwReason;
    v6 = fdwReason;
    LABEL_12:
    if ( v5 == v6 )
        return 0;
    if ( fdwReason -- 1 )
        CreateThread(0, 0, StartAddress, 0, 0, 0);
        return 1;
}
```
Case Study: “Sample J” Malware

**Sample J Call Trace Example**

```c
memset(, 0, 0x124u)
CreateToolhelp32Snapshot()
Process32First()
stricmp(., “explorer.exe”)
Process32Next(.,.)
stricmp(., “explorer.exe”)
CreateThread(,,StartAddress)
```

**Call Trace Analysis**

i. Iterate through the process handles to see if a process with name “explorer.exe” exists *(Check if user logged in)*

ii. If process exists, create a thread that infects target system
Questions:

1. How do we automate the call trace generation process in a manner that maximizes traversal of unique code paths?
2. How do we leverage generated call trace information to identify potential APTs?
PHASE II DETECTION

Automated Dynamic Call Trace Generation
Call Trace Generation

i. **Load** the dumped binary into a disassembler (e.g. IDA or Binary Ninja) and extract the executable portion of binary

ii. **Lift** the executable portion of binary to vex, a RISC-like intermediate representation (ir) language

iii. **Perform** emulation on the vex ir to traverse unique code paths that originate from a specified entry function

iv. **Record** calls that occur during traversal of a code path
AUTOMATING CALL TRACE GENERATION

main()

push ebp
mov rbp, rsp
call foo()

main() [vex ir]

t0 = GET:I64(bp)
t3 = GET:I64(rsp)
t2 = Sub64(t3, 0x008)
PUT(rsp) = t2
Stle(t1) = t0
t3 = GET:I64(rsp)
PUT(bp) = t3
call libfuncfoo()

t0 = GET:I64(bp)
t3 = GET:I64(rsp)
t2 = Sub64(t3, 0x008)
PUT(rsp) = t2
Stle(t1) = t0
t3 = GET:I64(rsp)
PUT(bp) = t3
call libfuncfoo()
(DEMO) CALL TRACE GENERATION

Target Binary: explorer.exe
(DEMO) CALL TRACE GENERATION
(DEMO) CALL TRACE GENERATION

INFO: FunctionCallGenerator: 19) [0x100093052] Name: 'GetWindowLongW' is_import='True' library_name: 'USER32' call_address: 0x1000bae748
INFO: FunctionCallGenerator: 20) [0x100093672] Name: 'SelectObject' is_import='True' library_name: 'GDI32' call_address: 0x1000ba748
INFO: FunctionCallGenerator: ========== END Call Trace ==========

INFO: FunctionCallGenerator: 1) [0x1000001e0] Name: 'sub_1000001e0' is_import='False' library_name: 'None' call_address: 0x1000001e0
INFO: FunctionCallGenerator: 2) [0x1000004ea9] Name: 'sub_1000004ea9' is_import='False' library_name: 'None' call_address: 0x1000004ea9
INFO: FunctionCallGenerator: 3) [0x10000046324] Name: 'SHIstChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x10000bb748
INFO: FunctionCallGenerator: 4) [0x10000040355] Name: 'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38
INFO: FunctionCallGenerator: 5) [0x1000006676a] Name: 'p_wm_QueServiceExec' is_import='True' library_name: 'SHLWAPI' call_address: 0x1000bab38
INFO: FunctionCallGenerator: 6) [0x1000006087] Name: 'sub_1000006087' is_import='False' library_name: 'None' call_address: 0x100006087
INFO: FunctionCallGenerator: 7) [0x10000054f1d] Name: 'GetDC' is_import='True' library_name: 'USER32' call_address: 0x100047814
INFO: FunctionCallGenerator: 8) [0x10000050bc] Name: 'sub_10000050bc' is_import='False' library_name: 'None' call_address: 0x1000050bc
INFO: FunctionCallGenerator: 9) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 10) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 11) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 12) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 13) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 14) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 15) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 16) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 17) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 18) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 19) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 20) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 21) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16
INFO: FunctionCallGenerator: 22) [0x10000050c16] Name: 'sub_10000050c16' is_import='False' library_name: 'None' call_address: 0x1000050c16

INFO: FunctionCallGenerator: ========== END Call Trace ==========

INFO: FunctionCallGenerator: 1) [0x1000001e0] Name: 'sub_1000001e0' is_import='False' library_name: 'None' call_address: 0x1000001e0
INFO: FunctionCallGenerator: 2) [0x1000004ea9] Name: 'sub_1000004ea9' is_import='False' library_name: 'None' call_address: 0x1000004ea9
INFO: FunctionCallGenerator: 3) [0x10000046324] Name: 'SHIstChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x10000bb748
INFO: FunctionCallGenerator: 4) [0x10000040355] Name: 'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38

INFO: FunctionCallGenerator: ========== END Call Trace ==========

INFO: FunctionCallGenerator: 1) [0x1000001e0] Name: 'sub_1000001e0' is_import='False' library_name: 'None' call_address: 0x1000001e0
INFO: FunctionCallGenerator: 2) [0x1000004ea9] Name: 'sub_1000004ea9' is_import='False' library_name: 'None' call_address: 0x1000004ea9
INFO: FunctionCallGenerator: 3) [0x10000046324] Name: 'SHIstChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x10000bb748
INFO: FunctionCallGenerator: 4) [0x10000040355] Name: 'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38

INFO: FunctionCallGenerator: ========== END Call Trace ==========

INFO: FunctionCallGenerator: 1) [0x1000001e0] Name: 'sub_1000001e0' is_import='False' library_name: 'None' call_address: 0x1000001e0
INFO: FunctionCallGenerator: 2) [0x1000004ea9] Name: 'sub_1000004ea9' is_import='False' library_name: 'None' call_address: 0x1000004ea9
INFO: FunctionCallGenerator: 3) [0x10000046324] Name: 'SHIstChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x10000bb748
INFO: FunctionCallGenerator: 4) [0x10000040355] Name: 'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38
Leverage Call Trace Info to Detect APTs

PHASE II DETECTION
**Leverage Call Trace Info to Detect APTs**

**Step 1:** Generate a vector representation of the set of call traces

Traces generated via *dynamic analysis*, where different code paths are traversed to maximize code coverage.

**Step 2:** Build a unified trusted call trace vector from trusted binaries

**Step 3:** Compare target binary against unified trusted call trace vector

Function output is percentage of call sequences in target that are trusted (1 ➞ all sequences trusted).
Step 1: Generate a vector representation of the set of call traces

- Call Trace 1
- Call Trace 2
- Call Trace n

Traces generated via dynamic analysis, where different code paths are traversed to maximize code coverage.

Step 2: Build a unified trusted call trace vector from trusted binaries

Trusted Call Trace vectors

Step 3: Compare target binary against unified trusted call trace vector

Target vector

Comparison Function

Function output is percentage of call sequences in target that are trusted (1 → all sequences trusted).
Step 1: Generate a vector representation of the set of call traces

Traces generated via dynamic analysis, where different code paths are traversed to maximize code coverage

Step 2: Build a unified trusted call trace vector from trusted binaries

Step 3: Compare target binary against unified trusted call trace vector

Function output is percentage of call sequences in target that are trusted (1 ➔ all sequences trusted)
Step 1: Generate a vector representation of the set of call traces

Traces generated via dynamic analysis, where different code paths are traversed to maximize code coverage

Step 2: Build a unified trusted call trace vector from trusted binaries

Step 3: Compare target binary against unified trusted call trace vector

Function output is percentage of call sequences in target that are trusted (1 ➞ all sequences trusted)

(See Appendix E for more details about the algorithm)
**CONCLUSION**

- **Phase I**: Provide us with a computationally efficient way to rapidly identify memory artifacts with anomalous code.

- **Phase II**: We leverage call trace information to differentiate between benign anomalous code and malware.
CONCLUSION

- Security is hard... Why not make it even harder for the adversary?

- Specifically, require the adversary to develop techniques to challenge this memory forensics approach that are both **reliable** (e.g. few bugs) and **portable** (e.g. works across various versions of the OS and binaries)


How different are equivalent types of memory artifacts (originating from identical binaries) across multiple hosts?

- The theory and the empirical results* suggest that memory artifacts are almost identical (+99% similar based on empirical results)
- *Under the following assumptions
  i. The hosts are not under significant memory pressure
  ii. Identical versions of the host operating system
Can we use hashes to determine if memory artifacts across hosts are identical (e.g. identical explorer.exe process artifacts on Hosts A & B)?

- **No.** Chunks of the binary may not be in memory because the OS has likely **paged** those sections to disk in order to efficiently utilize/manage memory.
- Also, the binary is likely **memory mapped**. Therefore, the OS may take a lazy approach by loading a particular chunk when needed.
- As a consequence, the binary on-disk will be different then its image that resides in memory. Memory artifacts across hosts are very likely to also be different.
Appendix

- Machine Learning Primer
- Advanced Binary Analysis
- Memory Forensics
- Binary Vector Generation
- Call Trace Vector Generation
- Hooking
APPENDIX A

Machine Learning Primer
Key Concepts

- **Unsupervised Learning**: Inferring a function to describe hidden structure from "unlabeled" data
- **Supervised Learning**: Inferring a function from *labeled training data*
- **Clustering**: Grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar
- **Classification**: Identifying to which of a set of categories (sub-populations) a new observation belongs
MACHINE LEARNING

- Challenges/Steps

1. **(Non-Trivial) Representing** observed/collected data in a meaningful mathematical expressions (e.g. vector)

2. **Deciding** on a metric for measuring the similarity of observations (e.g. Jaccard Similarity function)

3. **Selecting** a suitable algorithm that can classify and/or cluster observations appropriately using a supervised or unsupervised approach (e.g. agglomerative hierarchical clustering)
**Example**: Group squares based on similarity of the color types

- $S_1$
- $S_2$
- $S_3$
1. Representing Observations Mathematically

- We’ll represent the contents of each square as a vector where each dimension represents the number of occurrences of a color.

- Vector representations:
  - First square: [0, 0, 3]  
  - Second square: [1, 1, 1]  
  - Third square: [0, 2, 1]
2. Metric for measuring similarity of observations

- We’ll use the Jaccard Index to measure similarity

\[ J(S_i, S_j) = \frac{S_i \cap S_j}{S_i \cup S_j} = \frac{\# \text{ common color types}}{\# \text{ total color types}} \]

- Example:

\[ J(S_2, S_3) = \frac{S_2 \cap S_3}{S_2 \cup S_3} = \frac{2}{3} \]
3. Selecting a suitable algorithm

- We’ll use a hierarchical clustering algorithm
- Depending on the input parameters of the algorithm, the clusters could look like the following

\[ S_1 \quad S_2 \quad S_3 \]
**Binary Analysis**

- **Static Analysis**: Analysis of computer software that is performed without the actual execution of the software code.

- **Dynamic Analysis**: Execution of software in an instrumented or monitored manner to garner more concrete information on behavior.
**Binary Analysis**

- **Static vs. Dynamic Analysis**
  - **Static analysis** scales well and can provide better code coverage of a binary
  - **Dynamic analysis** can provide more accurate information on the actual execution behavior of a binary
  - **Static analysis** can produce false execution behavior as code paths may not be reachable during actual execution
  - **Dynamic analysis** can be computationally expensive
**Binary Analysis**

- Advanced analysis techniques
  - **Symbolic Execution**: Analysis of a program to determine the necessary inputs needed to reach a particular code path. Variables modeled as symbols
  - **Concolic Execution**: Used in conjunction with symbolic execution to generate concrete inputs (test cases) from symbolic variables to feed into program
  - **Selective Concolic Execution**: Selectively leverage concolic execution when fuzzing engine gets “stuck” (i.e. unable to generate inputs that can traverse a desired code path)
Motivational example for Symbolic Execution

```c
int main(void) {
    char buf[32];

    char *data = read_string();
    unsigned int magic = read_number();

    // difficult check for fuzzing
    if (magic == 0x31337987) {
        // Bad stuff
        doBadStuff();
    }
    else if(magic < 100 && magic % 15 == 2 && magic % 11 == 6) {
        // Only solution is 17;
        doReallyBadStuff();
    }
    else{
        doBenignStuff();
    }
}
### Motivational example for Symbolic Execution

```c
int main(void) {
    char buf[32];

    char *data = read_string();
    unsigned int magic = read_number();

    // difficult check for fuzzing
    if (magic == 0x31337987) {
        // Bad stuff
        doBadStuff();
    } else if (magic < 100 && magic % 15 == 2 && magic % 11 == 6) {
        // Only solution is 17;
        doReallyBadStuff();
    } else {
        doBenignStuff();
    }
}
```

Symbolic execution allows us to figure out the conditions (i.e. magic=0x31337987) to exercise this code path.

A more sophisticated code path that can be reached via symbolic execution.
Analyzing the call traces of example

- **Call Trace A**: Likely result if utilizing traditional emulation techniques to analyze sophisticated malware

- **Call Trace B**: The more useful trace for identifying potential malicious behavior of a binary as a result of applying advanced binary analysis techniques
**Memory Forensics**

- **Defined**: Analysis of a computer’s memory dump

- **Memory Acquisition**
  - Refers to the process of accessing the physical memory
  - *Most critical step in the memory forensics process*
  - Software and hardware tools can be used during acquisition, but we’ll focus on the former

- **Rekall** and **Lime** provide open source **acquisition** tools

- **Volatility** and **Rekall** provide open source **analysis** tools
Virtual Addressing and Memory Acquisition
Virtual Addressing and Memory Acquisition (cont’d)

- Physical space may be smaller than virtual address space.
- Less recently used memory blocks (a.k.a. pages) are moved to disk.
- **Important:** Only data that is in physical memory during acquisition can be acquired; paged data is unavailable.
Anti-Forensics

- Any attempt to compromise the availability or usefulness of evidence to the forensic process
- Techniques include
  
  i. **Substitution Attack**: Data fabricated by the attacker is substituted in place of valid data during the acquisition
  
  ii. **Disruption Attack**: Disrupt the acquisition process

- Proof of Concepts:
  
  i. “ShadowWalker” @ Blackhat 2005
  
  ii. “Low Down and Dirty” @ Blackhat 2006
  
  iii. “Defeating Windows Forensics” @ Fahrplan 2012

- The presented approach is resilient to anti-forensics techniques due to information asymmetry on the side of the defender
Generating a Vector Representation of a Binary
### Binary Vector Generation

- **Binary Vector Generation**
  
  i. **Load** the dumped binary into a disassembler (e.g. IDA or Binary Ninja) and extract the executable portion of binary
  
  ii. **Lift** the executable portion of binary to a RISC-like intermediate representation (ir)
  
  iii. **Create** a binary behavior report that categorizes each ir statement
  
  iv. **Generate** a set of fixed-sized linear sequences (w.r.t. address space) of ir statements that will be referred to as behavior sequences
  
  v. **Create** a map that maps each unique sequence to a row in a vector
**Binary Vector Generation**

**Disassembler**

**main()**
- push ebp
- mov rbp, rsp
- sub rsp, 0x10

**RISC-like Instructions**

- \( t_0 = \text{GET:I64}(bp) \)
- \( t_3 = \text{GET:I64}(rsp) \)
- \( t_2 = \text{Sub64}(t_3, 0x008) \)
- \( \text{PUT}(rsp) = t_2 \)
- \( \text{Stle}(t_1) = t_0 \)
- \( t_3 = \text{GET:I64}(rsp) \)
- \( \text{PUT}(bp) = t_3 \)
- \( t_4 = \text{GET:I64}(rsp) \)
- \( t_5 = 0x10 \)
- \( t_6 = \text{Sub64}(t_4, t_5) \)
- \( \text{PUT}(rsp) = t_6 \)

**Report**
- \( \text{reg_access} \)
- \( \text{reg_access} \)
- \( \text{arithmetic} \)
- \( \text{reg_access} \)
- \( \text{store} \)
- \( \text{reg_access} \)
- \( \text{reg_access} \)
- \( \text{reg_access} \)
- \( \text{other} \)
- \( \text{arithmetic} \)
- \( \text{reg_access} \)
# Binary Vector Generation

## Behavior Report

- `reg_access`
- `reg_access`
- `arithmetic`
- `reg_access`
- `store`
- `reg_access`
- `reg_access`
- `other`
- `arithmetic`
- `reg_access`

## Behavior Sequences

### SEQ 1

- `reg_access`
- `reg_access`
- `arithmetic`
**Binary Vector Generation**

- **Behavior Report**
  - `reg_access`
  - `reg_access`
  - `arithmetic`
  - `store`
  - `reg_access`
  - `reg_access`
  - `other`
  - `arithmetic`
  - `reg_access`

- **Behavior Sequences**
  - **SEQ 1**
    - `reg_access`
    - `reg_access`
    - `arithmetic`
  - **SEQ 2**
    - `reg_access`
    - `arithmetic`
    - `reg_access`
## Binary Vector Generation

<table>
<thead>
<tr>
<th>Report</th>
<th>SEQ 1</th>
<th>SEQ 2</th>
<th>SEQ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg_access</td>
<td>reg_access</td>
<td>reg_access</td>
<td>arithmetic</td>
</tr>
<tr>
<td>reg_access</td>
<td>reg_access</td>
<td>arithmetic</td>
<td>reg_access</td>
</tr>
<tr>
<td>arithmetic</td>
<td>arithmetic</td>
<td>reg_access</td>
<td>store</td>
</tr>
<tr>
<td>store</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arithmetic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>reg_access</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Binary Vector Generation**

**Behavior Report**
- reg_access
- reg_access
- arithmetic
- reg_access
- store
- reg_access
- reg_access
- reg_access
- other
- arithmetic
- reg_access

**Behavior Sequences**

**SEQ 1**
- reg_access
- reg_access
- arithmetic

**SEQ 2**
- reg_access
- arithmetic
- reg_access

**SEQ 3**
- arithmetic
- reg_access
- store

**SEQ 4**
- reg_access
- store
- reg_access
**Binary Vector Generation**

**Behavior Report**
- reg_access
- reg_access
- arithmetic
- reg_access
- store
- reg_access
- reg_access
- reg_access
- other
- arithmetic
- reg_access

**Behavior Sequences**

**SEQ 1**
- reg_access
- reg_access
- arithmetic

**SEQ 2**
- reg_access
- arithmetic
- reg_access

**SEQ 3**
- arithmetic
- reg_access
- store

**SEQ 4**
- reg_access
- store

**SEQ 5**
- store
- reg_access
- reg_access
- reg_access
**Binary Vector Generation**

**Behavior Report**

- reg_access
- reg_access
- arithmetic
- reg_access
- store

**Behavior Sequences**

**SEQ 1**
- reg_access
- reg_access
- arithmetic

**SEQ 2**
- reg_access
- arithmetic
- reg_access

**SEQ 3**
- arithmetic
- reg_access
- store

**SEQ 4**
- reg_access
- store

**SEQ 5**
- store
- reg_access
- reg_access

**SEQ 6**
- reg_access
- reg_access
- reg_access
**Binary Vector Generation**

- Total possible behavior sequence permutations: $\sim 2^{77}$
  - Number of instruction categories (e.g. store and branch): 14
  - Length of behavior sequence: 20
  - $14^{20} \sim 2^{77}$

- **Naive Approach**
  - Create a $14^{20}$ dimensional vector to express binary behavior
  - Each vector dimension maps to a unique behavior sequence
  - Number stored at dimension $k$ is the number of times behavior sequence occurs in executable
Naive Approach (continued…)

- Behavior Vector Generation

![Diagram showing a binary vector generation process.](image)
**Binary Vector Generation**

- **Naive Approach (continued...)**
  - Not very practical to implement directly
  - Fortunately, we can do better

- **Key observation**: Vector is sparse in that most of the dimensions will store the number ‘0’

- **Better Approach**
  - Only store the non-zero elements in memory
  - So if a binary has N total ir statements, then we only need to store at most $N - \text{window\_length}$ elements in memory
APPENDIX D

Efficient Differing Algorithm
### Efficient Diffing Algorithm

#### Step 2: Compute the similarity of a pair of vectors

<table>
<thead>
<tr>
<th>explorerA.exe</th>
<th>explorerB.exe</th>
<th>Similarity Function</th>
<th>.867</th>
</tr>
</thead>
</table>

**Similarity function takes as input two vectors and produces a value between 0 and 1**

### Selecting a similarity function

- For convenience, we’ll use the Jaccard Index
- The Jaccard Index has the following interpretation:

\[
J(S_i, S_j) = \frac{S_i \cap S_j}{S_i \cup S_j} = \frac{\text{#common behavior sequences}}{\text{#total behaviors sequences}}
\]

(See Appendix A for a simple example using the Jaccard Index)
APPENDIX E

Call Trace Vector Generation
Call Trace Vector Generation

- Call Trace Vector Generation
  
i. **Load** the dumped binary into a disassembler (e.g. IDA or Binary Ninja) and extract all sections of the binary
  
ii. **Lift** the executable portion of binary to a RISC-like intermediate representation (ir)
  
iii. **Perform** Dynamic analysis on the ir to generate call traces
  
iv. **Generate** a call trace vector that maps unique call trace sequences into a row of the vector
# Call Trace Vector Generation

## Call Trace (Code Path 1)

- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()

## Call Trace (Code Path 2)

- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()

## Call Trace (Code Path N)

- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()
CALL TRACE VECTOR GENERATION

Call Trace()

Trace Sequences

Call Trace (Code Path n)
main()
LoadLibraryW()
RegisterShellHook()
GetTokenInformation()
LsaOpenPolicy()
VirtualAlloc()
CreateEventW()
LogoffWindowsDialog()

Trace Seq 1
main()
LoadLibraryW()
RegisterShellHook()
CALL TRACE VECTOR GENERATION

Call Trace (Code Path n)
- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()

Trace Sequences

Trace Seq 1
- main()
- LoadLibraryW()
- RegisterShellHook()

Trace Seq 2
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
CALL TRACE VECTOR GENERATION

Call Trace (Code Path n)
- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()

Trace Sequences

Trace Seq 1
- main()
- LoadLibraryW()
- RegisterShellHook()

Trace Seq 2
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()

Trace Seq 3
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
CALL TRACE VECTOR GENERATION

Call Trace (Code Path n)
- main()
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
- CreateEventW()
- LogoffWindowsDialog()

Trace Sequences

Trace Seq 1
- main()
- LoadLibraryW()
- RegisterShellHook()

Trace Seq 2
- LoadLibraryW()
- RegisterShellHook()
- GetTokenInformation()

Trace Seq 3
- RegisterShellHook()
- GetTokenInformation()
- LsaOpenPolicy()

Trace Seq 4
- GetTokenInformation()
- LsaOpenPolicy()
- VirtualAlloc()
**CALL TRACE VECTOR GENERATION**

### Trace Call Vector

- **Index 0**
  - `main()`
  - `LoadLibraryW()`
  - `RegisterShellHook()`

- **Index \( k \)**
  - \( m \) occurrences of trace sequence \( k \)

- **Index \( 14^{20} - 1 \)**
  - `call`
  - `call`
  - `call`
  - `........`
  - `call`

- **Trace Call Vector**

- **Number of occurrences \( m \) of trace sequence \( k \)**
CALL TRACE VECTOR GENERATION

Trusted

Call Behavior Vector

Unknown

Host F
APPENDIX F

Hooking
**Monitoring Behavior w/ Hooks**

- Detouring a number of common APIs (e.g. CreateFile) to ensure monitoring code is executed before actual code.
- Depending on a set of rules (typically dynamic), the monitor code blocks, allows, or reports execution of API.