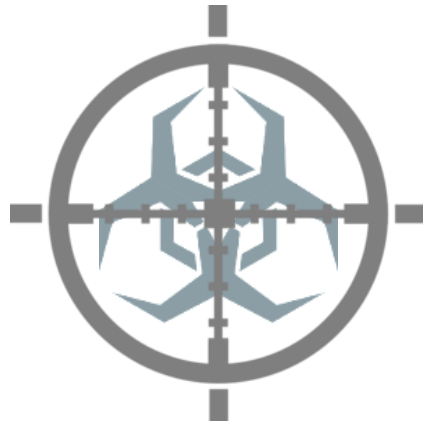


# Automated In-memory Malware/Rootkit Detection via Binary Analysis and Machine Learning

By: Malachi Jones, PhD



# ABOUT ME



<https://www.linkedin.com/in/malachijonesphd>



## ■ 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

- **PhD Thesis:** Asymmetric Information Games & Cyber Security (2009-2013)
- **Harris Corp.:** Cyber Software Engineer/ Vuln. Researcher (2013-2015)
- **Booz Allen Dark Labs:** Embedded Security Researcher (2016- Present)



# OUTLINE

- Motivation
- Memory Acquisition Automation
- Phase I Detection: Static Analysis
  - i. Efficiently Detecting Dissimilarities in Memory Artifacts
  - ii. Clustering Memory Artifacts at Scale to Detect Anomalies
- Phase II Detection: Dynamic Analysis
  - i. Automating Dynamic Call Trace Generation
  - ii. Leveraging Call Trace Information to Detect APTs
- Conclusion/ Q&A
- Appendix



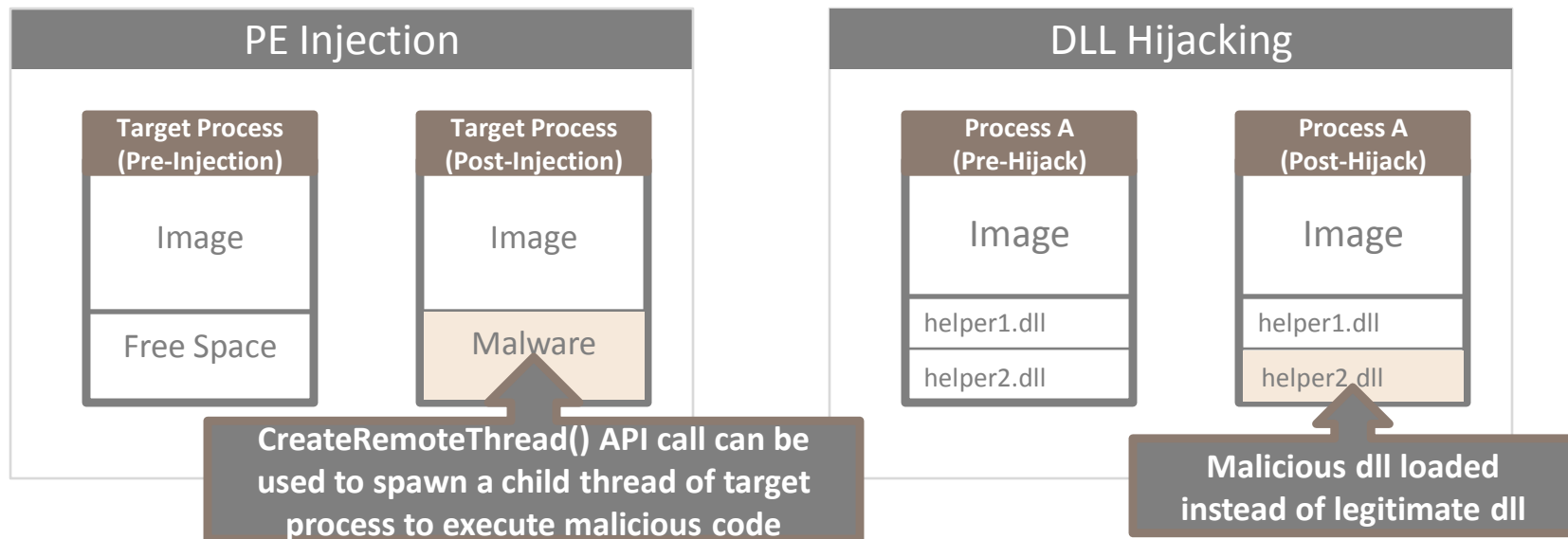
# APPENDIX

- Machine Learning Primer A
- Advanced Binary Analysis B
- Memory Forensics C
- Binary Vector Generation D
- Call Trace Vector Generation E
- Hooking F



# 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



# MOTIVATION



- **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*



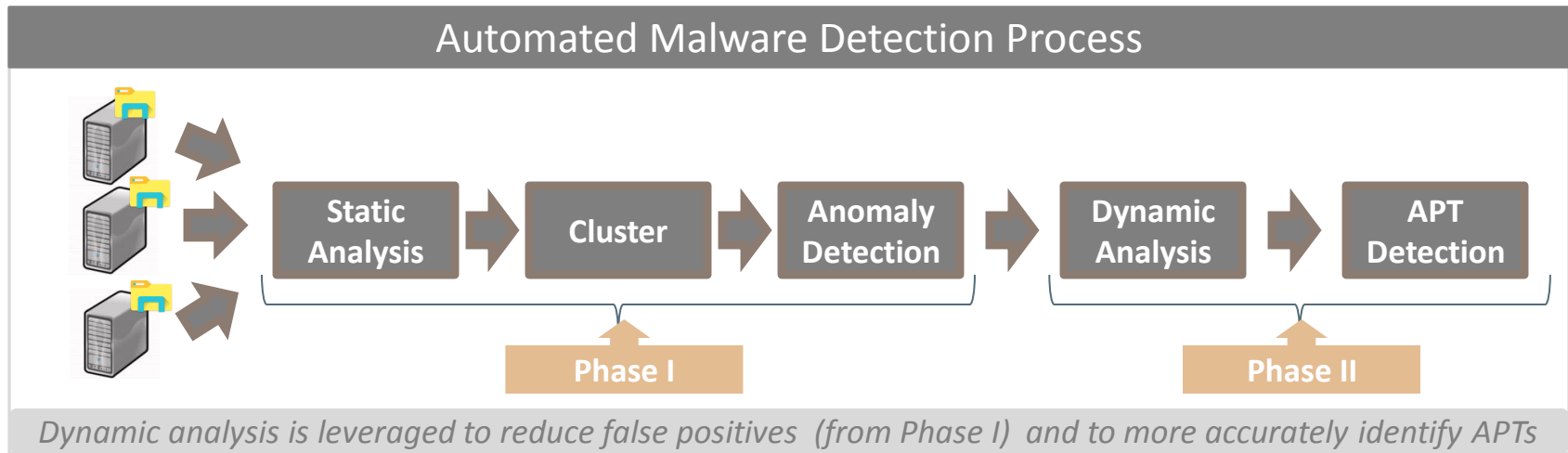


# OBJECTIVE

- **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



Host E



Host A



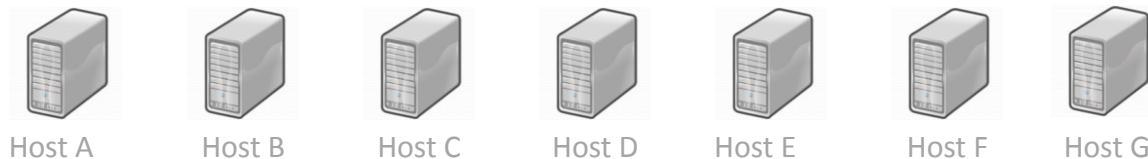
Host C

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



# MAIN TAKEAWAYS

## Set of Networked Hosts



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



## Memory Artifacts Sent to Collection Server



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



## Identical types of artifacts are clustered

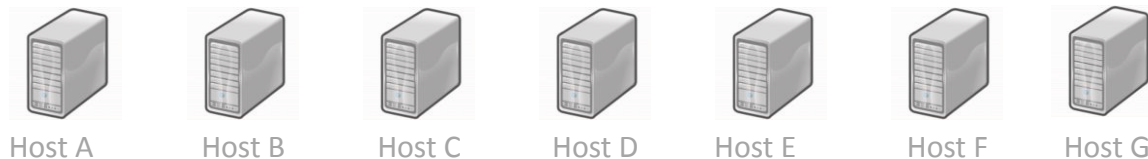


*Outliers (e.g. Host F) are analyzed dynamically to more accurately identify malicious behavior*



# MAIN TAKEAWAYS

## Set of Networked Hosts



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



## Memory Artifacts Sent to Collection Server



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



## Identical types of artifacts are clustered



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



# MAIN TAKEAWAYS

## Set of Networked Hosts



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



## Memory Artifacts Sent to Collection Server



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



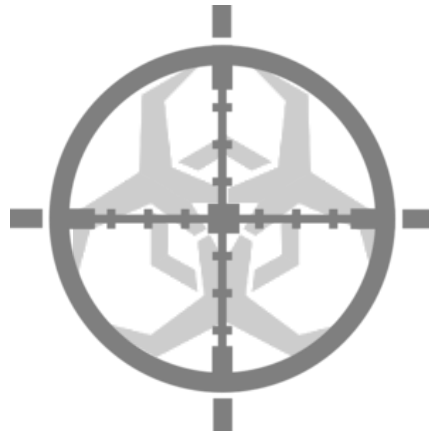
## Identical types of artifacts are clustered



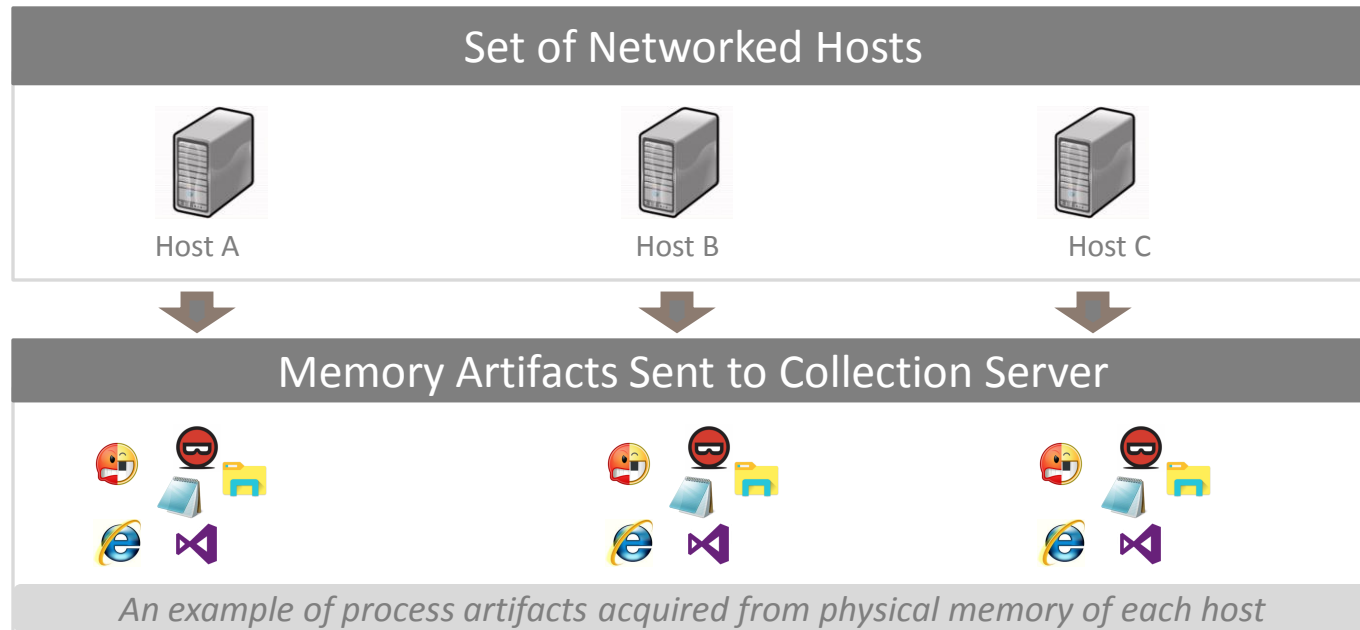
*Outliers (e.g. Host F) are analyzed dynamically to more accurately identify malicious behavior*



# MEMORY ACQUISITION AUTOMATION



# MEMORY ACQUISITION AUTOMATION



- **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





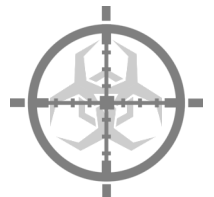
# MEMORY ACQUISITION AUTOMATION

- Querying for active and terminated processes

```
Administrator: C:\Windows\System32\cmd.exe - rekall -v live
[1] Live (Memory) 16:20:15> pslist
-----> pslist()
2017-08-30 16:20:15,434:DEBUG:rekall.1:Running plugin (pslist) with args (()) kwargs ({})
```

_EPROCESS	name	pid	ppid	thread_count	handle_count	session_id	wow64	process_create_time	process_exit_time
0xfa800c728040	System	4	0	132	1194	-	False	2017-08-02 19:18:50Z	-
0xfa800d9deb10	smss.exe	324	4	2	32	-	False	2017-08-02 19:18:50Z	-
0xfa80106af8e0	notepad++.exe	332	2732	0	-	1	False	2017-08-04 01:54:03Z	2017-08-04 01:54:03Z
0xfa800ef96060	stacsv64.exe	368	660	13	213	0	False	2017-08-02 19:18:55Z	-
0xfa800d663460	AmazonDrive.ex	380	6628	1	149	1	True	2017-08-07 12:56:47Z	-
0xfa800ef7f060	svchost.exe	392	660	48	2000	0	False	2017-08-02 19:18:55Z	-
0xfa800e96fb10	csrss.exe	452	388	10	1011	0	False	2017-08-02 19:18:54Z	-
0xfa800ef13510	svchost.exe	460	660	23	801	0	False	2017-08-02 19:18:55Z	-

*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

```
Administrator: C:\Windows\System32\cmd.exe - rekall -v live
[1] Live (Memory) 16:20:15> pslist
-----> pslist()
2017-08-30 16:20:15,434:DEBUG:rekall.1:Running plugin (pslist) with args (()) kwargs ({})
```

_EPROCESS	name	pid	ppid	thread_count	handle_count	session_id	wow64	process_create_time	process_exit_time
0xfa800c728040	System	4	0	132	1194	-	False	2017-08-02 19:18:50Z	-
0xfa800d9deb10	smss.exe	324	4	2	32	-	False	2017-08-02 19:18:50Z	-
0xfa80106af8e0	notepad++.exe	332	2732	0	-	1	False	2017-08-04 01:54:03Z	2017-08-04 01:54:03Z
0xfa800ef96060	stacsv64.exe	368	660	13	-	0	False	2017-08-02 19:18:55Z	-
0xfa800d663460	AmazonDrive.exe	380	6628	1	-	1	True	2017-08-07 12:56:47Z	-
0xfa800ef7f060	svchost.exe	392	-	-	-	-	-	02 19:18:55Z	-
0xfa800e96fb10	csrss.exe	452	-	-	-	-	-	02 19:18:54Z	-
0xfa800ef13510	svchost.exe	460	-	-	-	-	-	02 19:18:55Z	-

Terminated process  
(notepad++) still in memory  
26 days later

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



# MEMORY ACQUISITION AUTOMATION

- Capturing live dumps of binaries w/ Rekall

```
[1] Live (Memory) 16:35:15> procdump pids=[332,324,368,380], dump_dir="C:/tmp/dumps"  
-----> procdump(pids=[332,324,368,380], dump_dir="C:/tmp/dumps")  
2017-08-30 16:35:15,786:DEBUG:rekall.1:Running plugin (procdump) with args (()) kwargs  
'}')  
  
      _EPROCESS      Filename  
-----  
0xfa800d9deb10 smss.exe      324 executable.smss.exe_324.exe  
0xfa80106af8e0 notepad++.exe  332 executable.notepad.exe_332.exe  
0xfa800ef96060 stacsv64.exe  368 executable.stacsv64.exe_368.exe  
0xfa800d663460 AmazonDrive.ex  380 executable.AmazonDrive.ex_380.exe  
Out<16:35:17> Plugin: procdump (ProcExeDump)  
[1] Live (Memory) 16:35:17>
```

*procdump dumps a set of specified processes (given pids as input) to a desired directory*



# MEMORY ACQUISITION AUTOMATION

- Capturing live dumps of binaries w/ Rekall

The image shows a terminal window on the left and a Windows File Explorer window on the right. The terminal window displays the output of a memory acquisition process, including a list of memory addresses and process names. The File Explorer window shows a directory named 'dumps' containing several executable files.

**Terminal Output:**

```
[1] Live (Memory) 16:35:15
-----
2017-08-30 16:35:15,786:D
umps' })
-----
_EPRO
-----
0xfa800d9deb10 smss.exe
0xfa80106af8e0 notepad++.exe
0xfa800ef96060 stacsv64.exe
0xfa800d663460 AmazonDrive
Out<16:35:17> Plugin: pro
[1] Live (Memory) 16:35:17
```

**File Explorer Window:**

Path: Computer > Local Disk (C:) > tmp > dumps

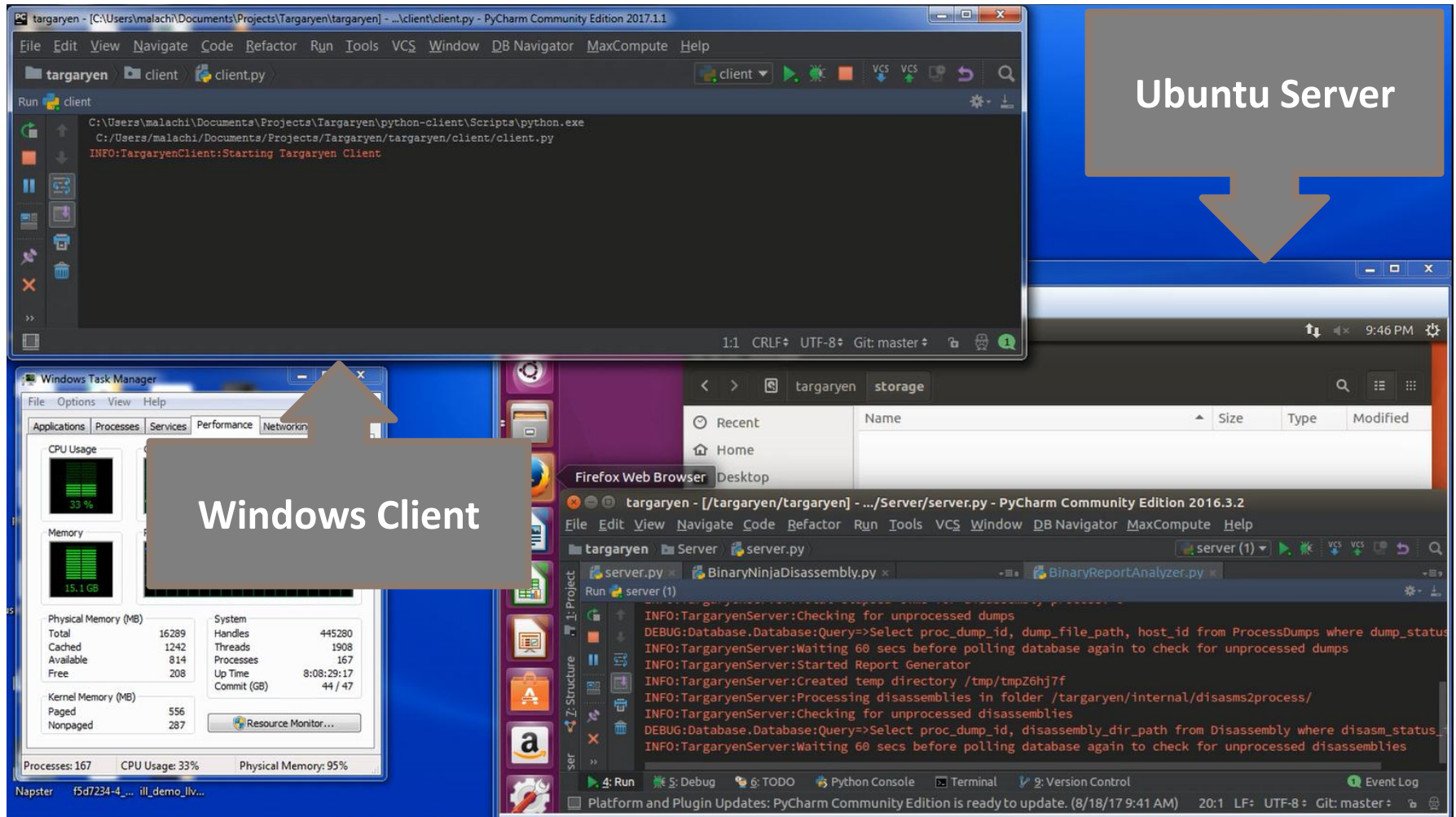
Name	Date modified	Type	Size
executable.AmazonDrive.ex_380	8/30/2017 4:35 PM	Application	4,757 KB
executable.notepad.exe_332	8/30/2017 4:35 PM	Application	977 KB
executable.smss.exe_324	8/30/2017 4:35 PM	Application	110 KB
executable.stacsv64.exe_368	8/30/2017 4:35 PM	Application	332 KB

Details for executable.notepad.exe\_332: Date modified: 8/30/2017 4:35 PM, Date created: 8/30/2017 4:34 PM, Size: 976 KB

*procdump dumps a set of specified processes (given pids as input) to a desired directory*



# (DEMO) MEMORY ACQUISITION AUTOMATION





# (DEMO) MEMORY ACQUISITION AUTOMATION

The image shows a Windows desktop environment with a PyCharm IDE running a client script. The client script is named `client.py` and is located in the `client` directory of the `targaryen` project. The script is running, and the output shows the client starting. A large arrow points from the Ubuntu server to the Windows client.

**Windows Client**

Windows Task Manager Performance tab:

CPU Usage	
33 %	

Memory	
15.1 GB	

Physical Memory (MB)	
Total	16289
Cached	1242
Available	814
Free	208

System	
Handles	445280
Threads	1908
Processes	167
Up Time	8:08:29:17
Commit (GB)	44 / 47

Kernel Memory (MB)	
Paged	556
Nonpaged	287

Processes: 167 CPU Usage: 33% Physical Memory: 95%

**Ubuntu Server**

PyCharm IDE running `server.py` on the Ubuntu server. The output shows the server checking for unprocessed dumps, waiting 60 seconds, and starting the report generator.

```
INFO:TargaryenServer:Checking for unprocessed dumps
DEBUG:Database.Database:Query=>Select proc_dump_id, dump_file_path, host_id from ProcessDumps where dump_status
INFO:TargaryenServer:Waiting 60 secs before polling database again to check for unprocessed dumps
INFO:TargaryenServer:Started Report Generator
INFO:TargaryenServer:Created temp directory /tmp/tmpZ6hj7f
INFO:TargaryenServer:Processing disassemblies in folder /targaryen/internal/disasms2process/
INFO:TargaryenServer:Checking for unprocessed disassemblies
DEBUG:Database.Database:Query=>Select proc_dump_id, disassembly_dir_path from Disassembly where disasm_status
INFO:TargaryenServer:Waiting 60 secs before polling database again to check for unprocessed disassemblies
```



# (DEMO) MEMORY ACQUISITION AUTOMATION

Process list queried and sent to server

Time	Process Name	PID	PPID	Session ID	Is System	
2017-08-29 21:36:43Z	calc.exe	13004	2732	4	83	1 False
2017-08-25 22:43:40Z	RdrCEF.exe	13064	12488	13	232	1 True
2017-08-29 15:36:01Z	calc.exe	13256	2732	4	80	1 False

Server processing client's process list

DEBUG:Database.Database:Added process 'calc.exe' with ID '11' into database  
DEBUG:Database.Database:Adding process 'services.exe' into database  
DEBUG:Database.Database:Query->INSERT INTO Processes(process\_name) VALUES (%s)  
DEBUG:Database.Database:Added process 'services.exe' with ID '12' into database  
DEBUG:Database.Database:Adding process 'winlogon.exe' into database  
DEBUG:Database.Database:Query->INSERT INTO Processes(process\_name) VALUES (%s)  
DEBUG:Database.Database:Added process 'winlogon.exe' with ID '13' into database  
DEBUG:Database.Database:Adding process 'lsass.exe' into database  
DEBUG:Database.Database:Query->INSERT INTO Processes(process\_name) VALUES (%s)



# (DEMO) MEMORY ACQUISITION AUTOMATION

The screenshot displays a PyCharm IDE environment with a client.py script running in a virtual machine (Ubuntu 64-bit 16\_04\_Targaryen). The script is outputting logs for RekallMemoryForensics, showing the dumping of various processes. A Windows Task Manager overlay is visible, showing system performance metrics. A file explorer window shows the directory /targaryen/storage/binaries/client\_malachi\_pc\_1, which contains several dumped process files. A text box with an arrow points to these files, stating "Dumped processes received from client".

PyCharm Run Output:

```
INFO:RekallMemoryForensics:Waiting on command to complete:'-----'
INFO:RekallMemoryForensics:Waiting on command to complete:'0xfa800f4dfb10 HeciServer.exe' 1176 executable.HeciServer.exe_1176.exe
INFO:RekallMemoryForensics:Waiting on command to complete:'0xfa80177c4160 DeviceDisplay0' 1204 executable.DeviceDisplay0_1204.exe
INFO:RekallMemoryForensics:Waiting on command to complete:'0xfa800f0fbb10 igfxCUIService' 1268 executable.igfxCUIService_1268.exe
INFO:RekallMemoryForensics:Waiting on command to complete:'0xfa800f130a40 DisplayLinkMan' 1348 executable.DisplayLinkMan_1348.exe
INFO:root:Sending response message (cmd_id=5) of type MessageTy
INFO:Common.TargaryenUtils:Total sent: 3756726 Data Length: 3
INFO:Common.TargaryenUtils:Total sent: 5 Data Length: 5
```

Windows Task Manager Performance:

- CPU Usage: 42%
- Memory: 15.5 GB
- Physical Memory (MB): Total 16289, Cached 1244, Available 405, Free 16
- System: Handles 447349, Threads 2075, Processes 175, Up Time 8:08:30:42, Commit (GB) 44 / 47
- Kernel Memory (MB): Paged 562, Nonpaged 287

File Explorer (client\_malachi\_pc\_1):

Name	Size	Type	Modified
1504144056_executable_AmazonDrive.ex_380...	1.0 MB	Archive	21:47
1504144056_executable_smss.exe_324.dmp.z	112.6 kB	Archive	21:47
1504144056_executable_stacsv64.exe_368.dmp.z	339.5 kB	Archive	21:47
1504144056_executable_svchost.exe_392.dmp.z	27.1 kB	Archive	21:47
1504144056_executable_System_4.dmp.z	1.0 MB	Archive	21:47
1504144057_executable_csrss.exe_452.dmp.z	1.0 MB	Archive	21:47
1504144057_executable_notepad.exe_480.dmp.z	193.5 kB	Archive	21:47
1504144057_executable_svchost.exe_460.dmp.z	27.1 kB	Archive	21:47

PyCharm Run Output (continued):

```
INFO:TargaryenServer:Storing proc dump 'mssec.exe' to filepath /targaryen/storage/binaries/client_malachi_pc_1
INFO:TargaryenServer:Storing proc dump 'svchost.exe' to filepath /targaryen/storage/binaries/client_malachi_pc_1
DEBUG:Database.Database:Query=>Select p.pid, s.process_info_id From SendProcDumpCmdInfo s, ProcessesInfo p WHERE
DEBUG:Database.Database:Query=> Update HostCommands SET command_status_id = 3 WHERE command_id = 5 AND host_id
INFO:TargaryenServer:Client client_malachi_pc_1 gracefully ended connection
```





# (DEMO) MEMORY ACQUISITION AUTOMATION

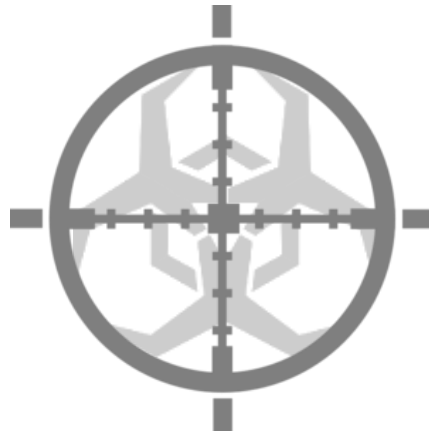
The screenshot illustrates a memory acquisition automation workflow. It features three main windows:

- PyCharm Community Edition 2017.1.1:** The top window shows the `client.py` script. The console output indicates that the `procdump` plugin is running with arguments for specific PIDs and a dump directory. The output shows the process list and the start of the dump process.
- Windows Task Manager:** The bottom-left window shows the system's resource usage. The CPU usage is 68%, and the physical memory usage is 97%. The system information section shows 16289 MB of total physical memory, with 1243 MB cached and 384 MB available.
- Disassembler (Binary Ninja):** The bottom-right window shows the disassembler interface. The file list on the right shows two files: `2_1_1504144056_executable_smss.exe_324` and `38_1_1504144086_executable_EvtEng.exe_2032`. The console output shows the disassembly process for the second file, indicating that the `BinaryNinja` disassembler is being used.

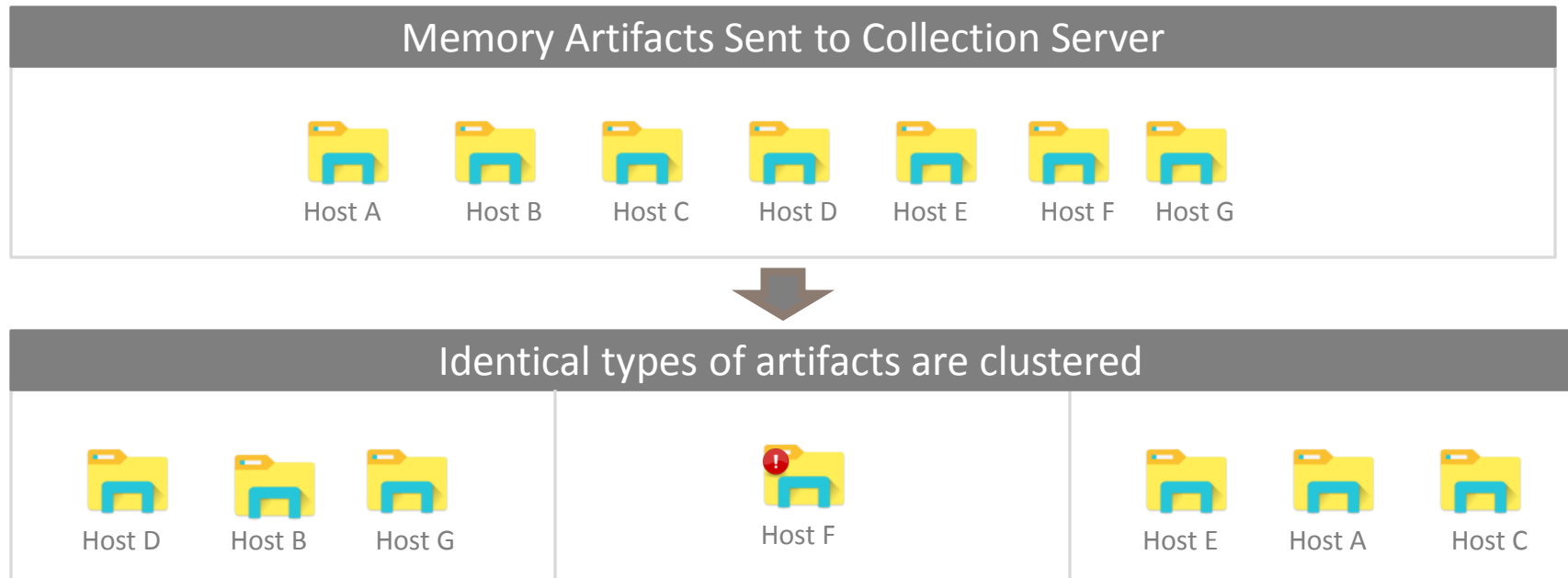
A text box with an arrow points to the disassembler window, stating: "Process dumps disassembled w/ Binary Ninja".



# PHASE I DETECTION: STATIC ANALYSIS



# PHASE I DETECTION: STATIC ANALYSIS

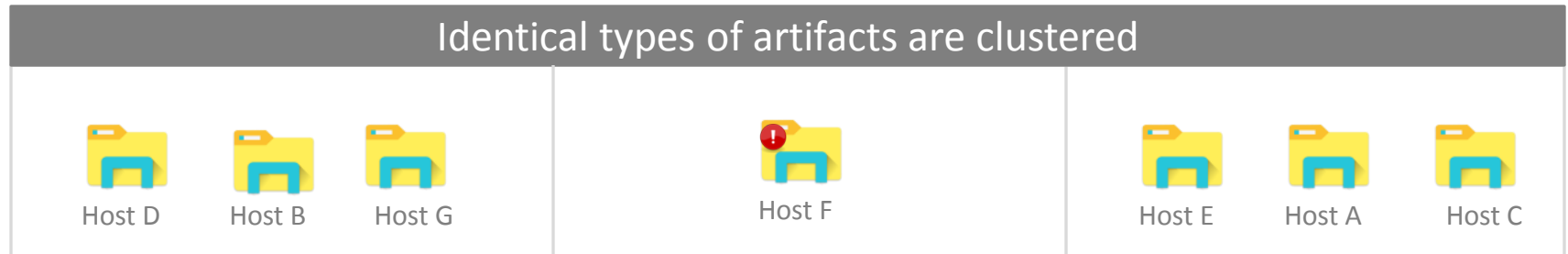


## ■ Overall Goals:

- *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*



# PHASE I DETECTION: STATIC ANALYSIS

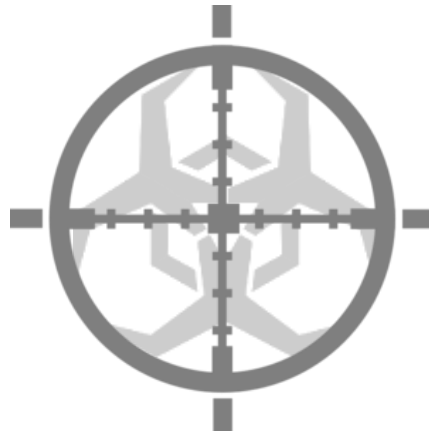


- **Requirements for clustering artifacts *at scale***
  - **Computationally efficient** method for determining the similarity (or dissimilarity) of a pair of binaries
  - Clustering algorithm with a **linear run-time performance in the worst-case**



Computationally Efficient Diffing Algorithm

# PHASE I DETECTION: STATIC ANALYSIS



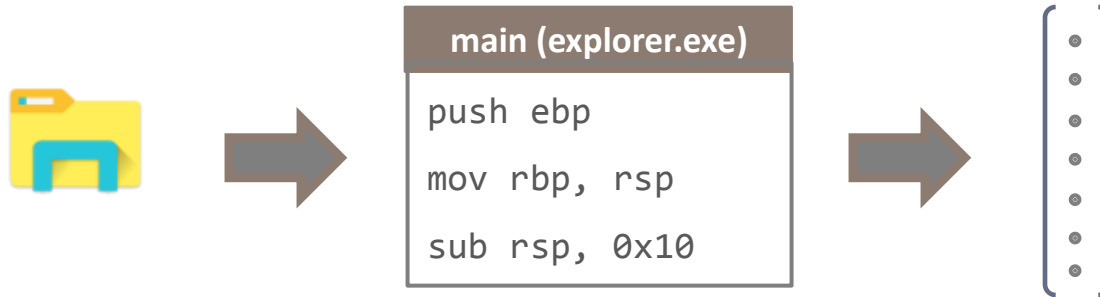
# EFFICIENT DIFFING ALGORITHM

- **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



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

(See Appendix D for more details about the algorithm)



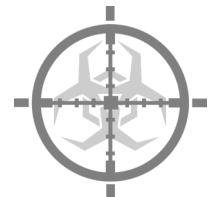
# EFFICIENT DIFFING ALGORITHM

- Evaluating similarity function of on-disk copy of explorer.exe vs. in-memory image

Name	Size	Type	Modified
 executable.1844.exe	2.9 MB	Program	Jun 2
 executable.1844.exeBinaryReport.xml	133.4 MB	Markup	May 15
 executable.1844.exeDisassembly.xml	26.0 MB	Markup	Jun 2
 explorer_memory_forensics.exe	2.9 MB	Program	Jun 2
 explorer_memory_forensics.exeBinaryReport.xml	178.6 MB	Markup	May 16
 explorer_memory_forensics.exeDisassembly.xml	34.4 MB	Markup	Jun 2

**In-Memory Copy** (points to executable.1844.exe)

**On-Disk Copy** (points to explorer\_memory\_forensics.exe)





# EFFICIENT DIFFING ALGORITHM

- Evaluating similarity function against explorer.exe

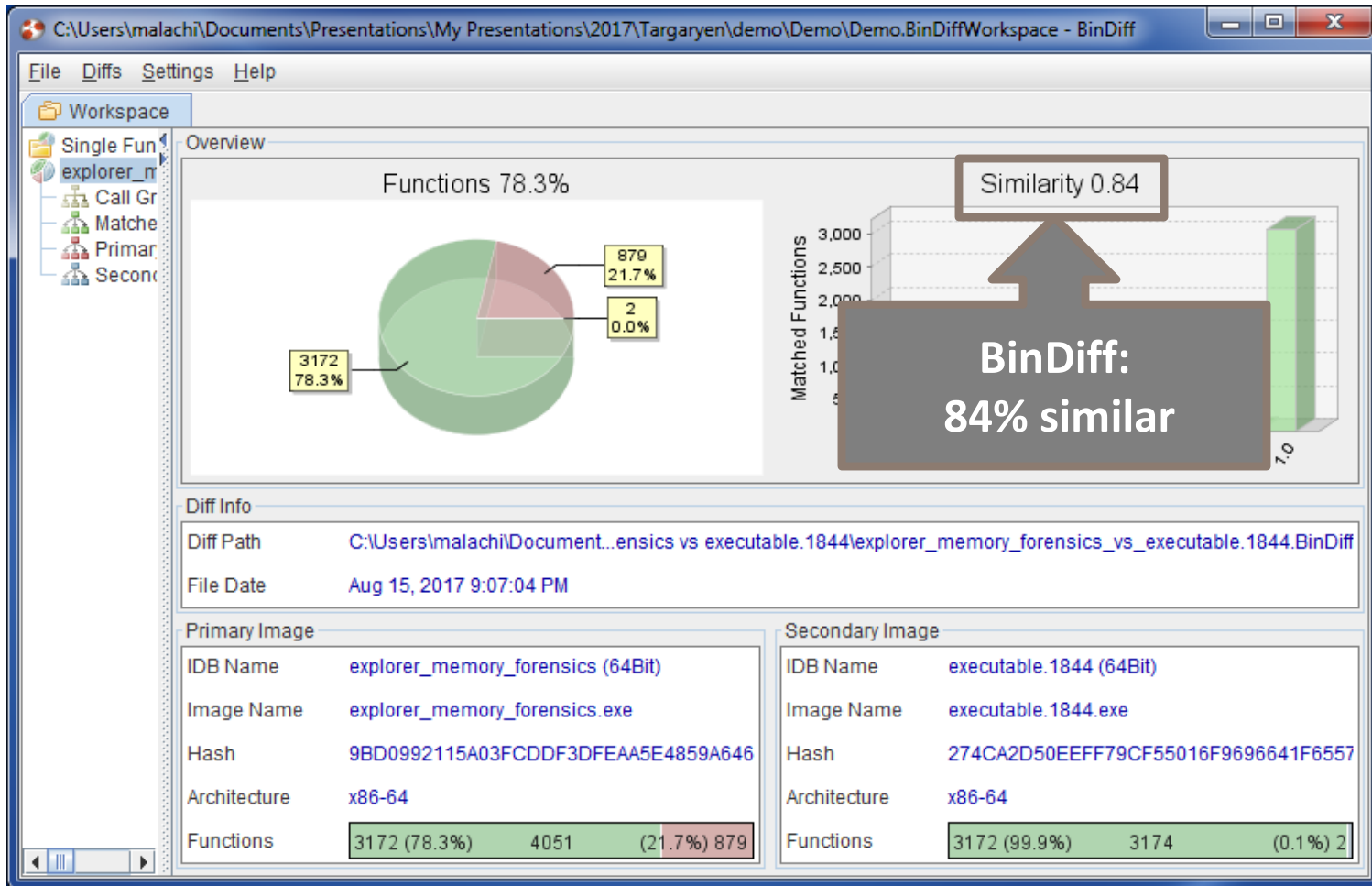
```
/home/targaryen/.virtualenvs/angr/bin/python /targaryen/targaryen/BinaryReportAnalyzer/ReportSimilarityAnalysis.  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:  
Report Similarity Analysis  
  
INFO:BinaryReportAnalyzer.ReportAnalysis:  
(Elapsed Time: 0.0248849391937) Deserialized report analysis  
  file 'executable.1844.exe'  
  hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a6287044d17690a75bf628'  
  
INFO:BinaryReportAnalyzer.ReportAnalysis:  
(Elapsed Time: 0.0328030586243) Deserialized report analysis  
  file 'explorer_memory_forensics.exe'  
  hash:'6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef9dab53aeff1cff0'  
  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Beginning Jacard index analysis  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Finished Jacard index analysis (Elapsed time:0.0579369068146)  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jacard index: 0.834103965252  
  
Process finished with exit code 0
```

Jaccard Index:  
83.4% similar



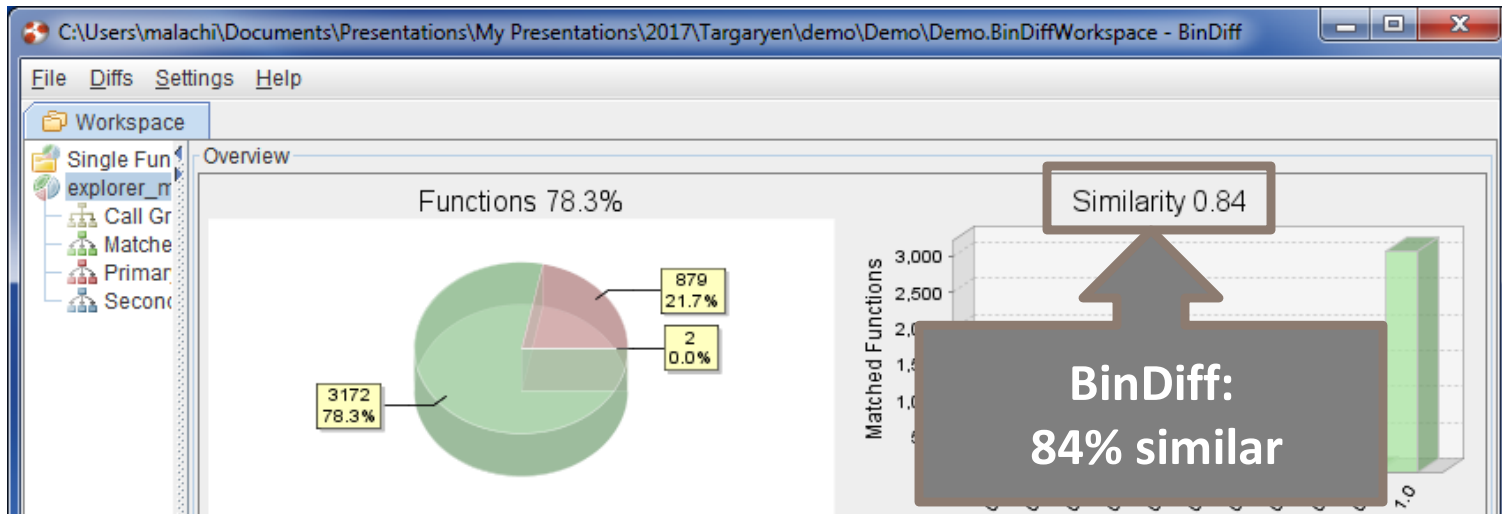
# EFFICIENT DIFFING ALGORITHM

- Comparing similarity results against BinDiff



# EFFICIENT DIFFING ALGORITHM

- Comparing similarity results against BinDiff



```
INFO:BinaryReportAnalyzer.ReportAnalysis:  
(Elapsed Time: 0.0248849391937) Deserialized report analysis  
file 'executable.1844.exe'  
hash: 'adaca26eb1685da66c547e01363664cc3bf38c2a08a628704'
```

```
INFO:BinaryReportAnalyzer.ReportAnalysis:  
(Elapsed Time: 0.0328030586243) Deserialized report analysis  
file 'explorer_memory_forensics.exe'  
hash: '6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef'
```

```
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Beginning Jacard index analysis  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Finished Jacard index analysis (Elapsed time: 0.000103965252)  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jacard index: 0.834103965252
```

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: '6bed1a3a956a859ef4420feb2466c040800eaf01ef53214ef9dab53aeff1cff0'
```

```
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Beginning Jacard index analysis  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Finished Jacard index analysis (Elapsed time:0.0579369068146)  
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:Jacard index: 0.834103965252
```

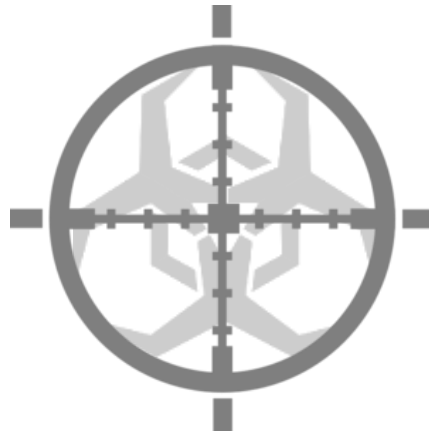
Jaccard Performance:  
57 ms

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



Clustering Memory Artifacts at Scale

# PHASE I DETECTION: STATIC ANALYSIS



# CLUSTERING ARTIFACTS AT SCALE

## ■ 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  $k \ll n$



# CLUSTERING ARTIFACTS AT SCALE

## ■ Approximate Clustering Algorithm [1] Sketch

Set of memory artifacts

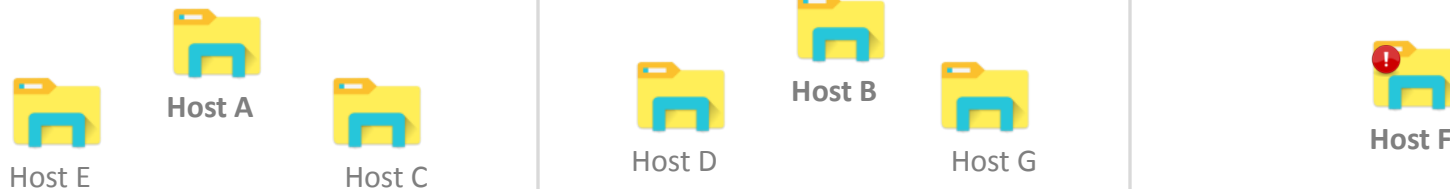


Step 1: Prototype Extraction [ $O(k \cdot n)$ ]



*Prototypes are a small ( $k \ll n$ ), yet representative subset of artifacts*

Step 2: Clustering w/ Prototype [ $O(k^2 \log(k) + n)$ ]



# CLUSTERING ARTIFACTS AT SCALE

## ■ Prototype Extraction Algorithm [1]

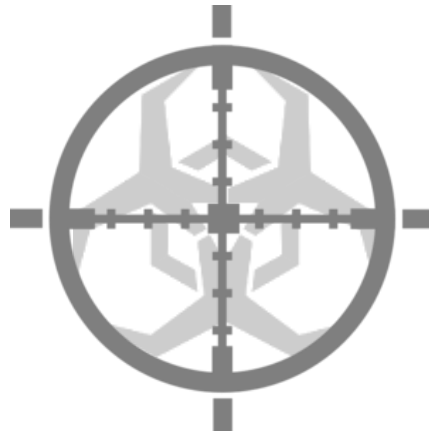
### Algorithm 1 Prototype extraction

```
1:  $prototypes \leftarrow \emptyset$ 
2:  $distance[x] \leftarrow \infty$  for all  $x \in reports$ 
3: while  $\max(distance) > d_p$  do
4:   choose  $z$  such that  $distance[z] = \max(distance)$ 
5:   for  $x \in reports$  and  $x \neq z$  do
6:     if  $distance[x] > ||\hat{\phi}(x) - \hat{\phi}(z)||$  then
7:        $distance[x] \leftarrow ||\hat{\phi}(x) - \hat{\phi}(z)||$ 
8:   add  $z$  to  $prototypes$ 
```

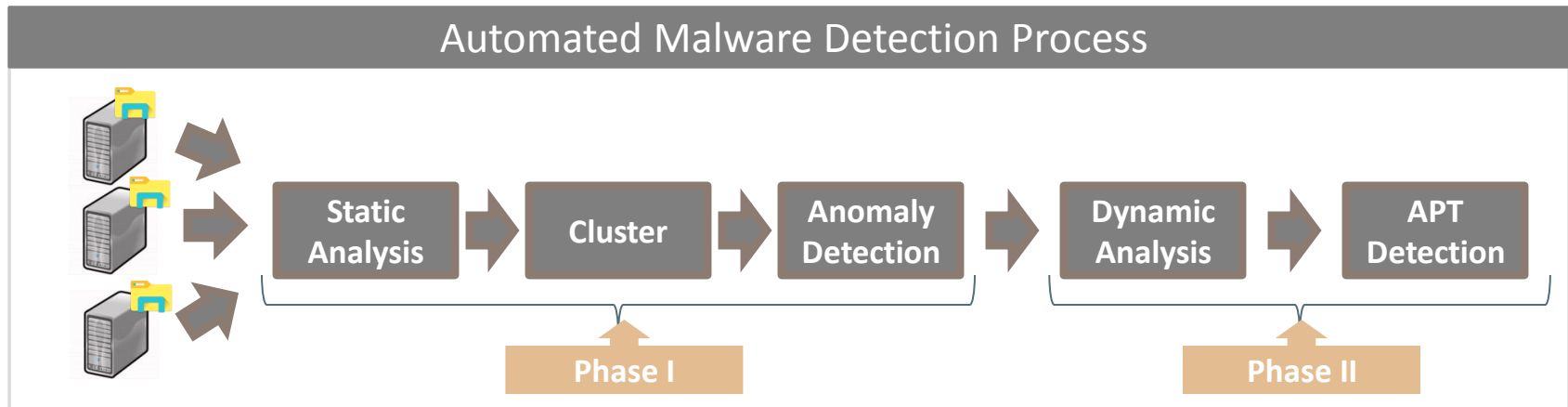




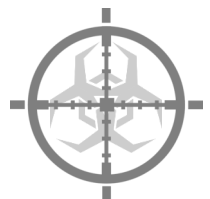
# PHASE II DETECTION: DYNAMIC ANALYSIS



# PHASE II: 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***



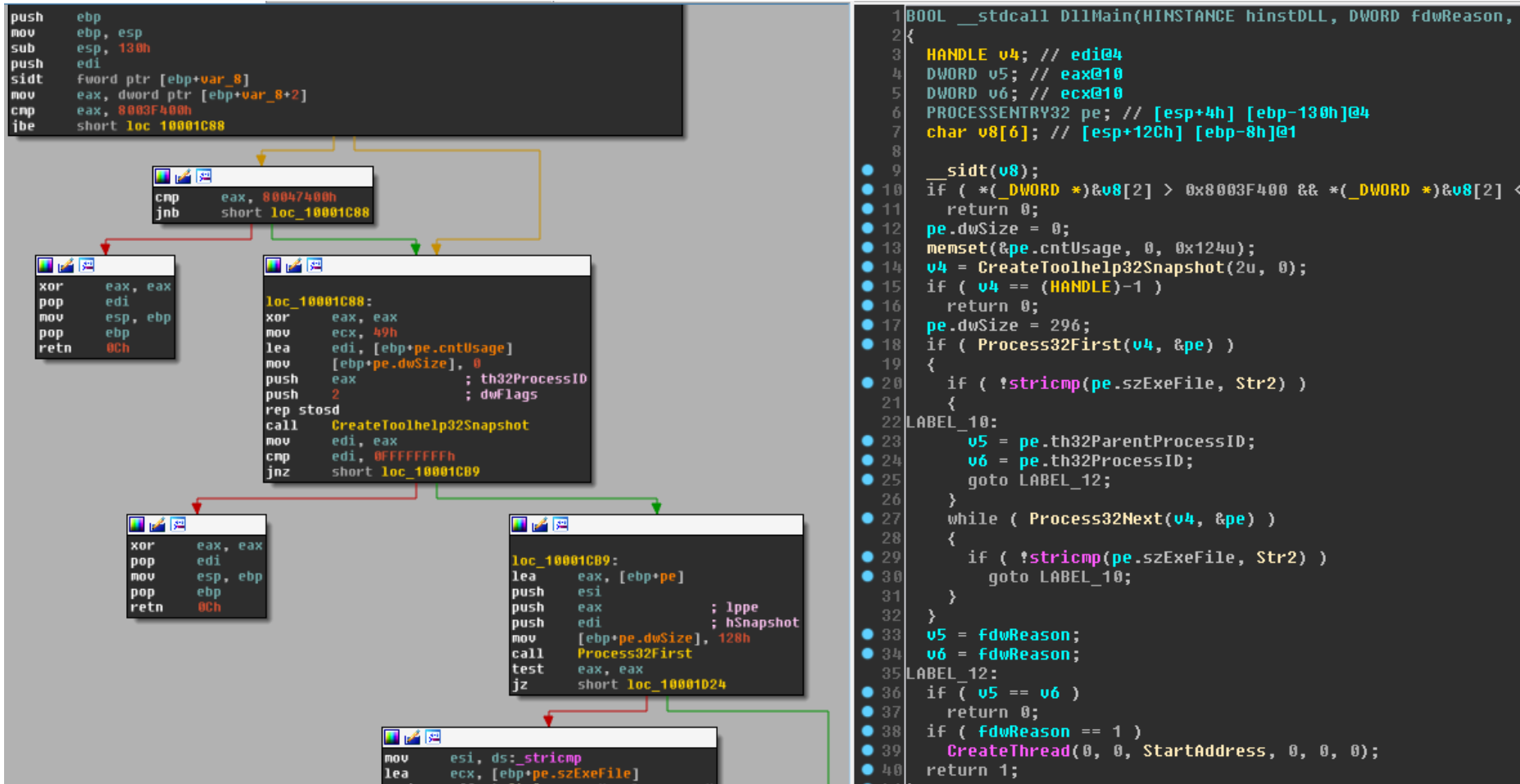
# PHASE II: DYNAMIC ANALYSIS

- **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*



# PHASE II: DYNAMIC ANALYSIS

## Case Study: “Sample J” Malware



(sha1: 70cb0b4b8e60dfed949a319a9375fac44168ccbb)

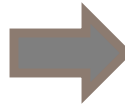


# PHASE II: DYNAMIC ANALYSIS

## Case Study: “Sample J” Malware

```
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] > 0x8003F400 && *(_DWORD *)&v8[2] <
        return 0;
    pe.dwSize = 0;
    memset(&pe.cnUsage, 0, 0x124u);
    v4 = CreateToolhelp32Snapshot(2u, 0);
    if ( v4 == (HANDLE)-1 )
        return 0;
    pe.dwSize = 296;
    if ( Process32First(v4, &pe) )
    {
        if ( !strcmp(pe.szExeFile, Str2) )
        {
            LABEL_10:
            v5 = pe.th32ParentProcessID;
            v6 = pe.th32ProcessID;
            goto LABEL_12;
        }
        while ( Process32Next(v4, &pe) )
        {
            if ( !strcmp(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;
}
```



### Sample J Call Trace Example

```
memset(,0, 0x124u)
CreateToolhelp32Snapshot()
Process32First()
strcmp(., "explorer.exe")
Process32Next(.,.)
strcmp(., "explorer.exe")
CreateThread(.,StartAddress)
```



# PHASE II: DYNAMIC ANALYSIS

## Case Study: “Sample J” Malware

### Sample J Call Trace Example

```
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



# PHASE II: DYNAMIC ANALYSIS

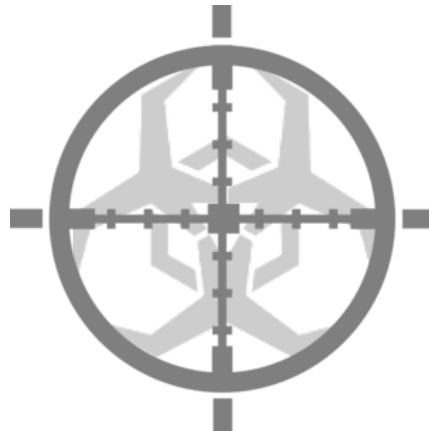
## ■ Questions:

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?



Automated Dynamic Call Trace Generation

## PHASE II DETECTION





# AUTOMATING CALL TRACE GENERATION

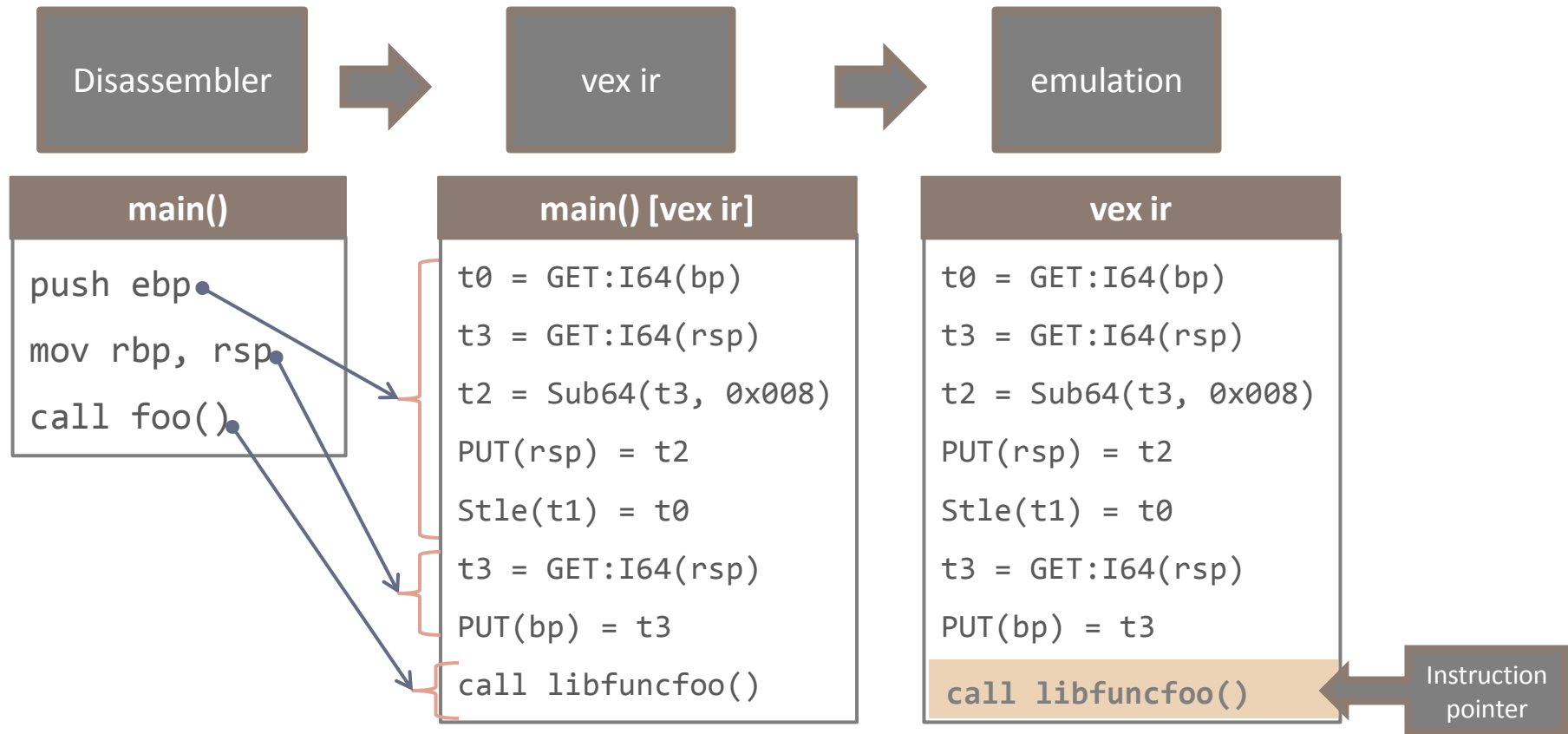


## ■ Call Trace Generation

- 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

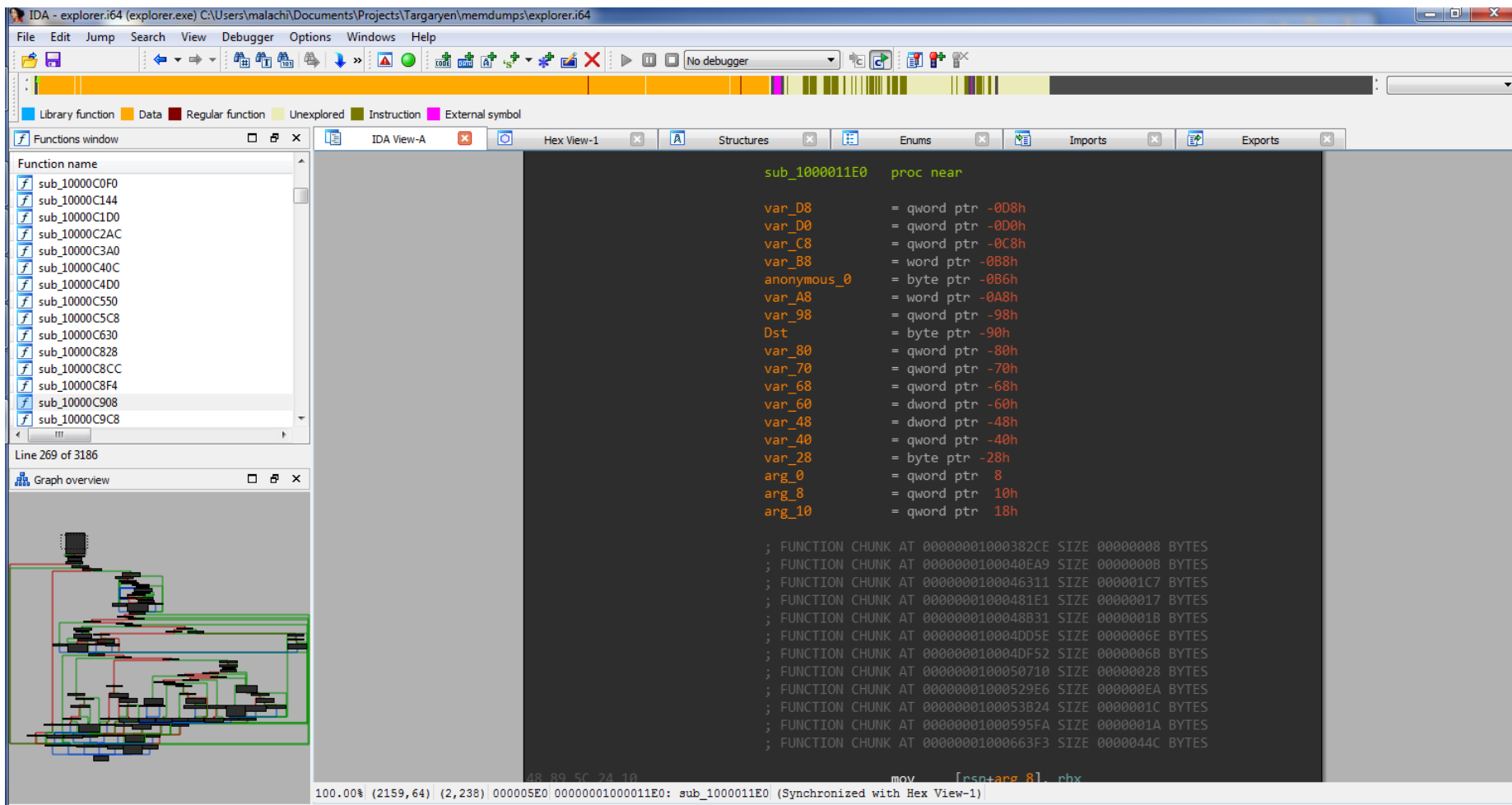


# AUTOMATING CALL TRACE GENERATION

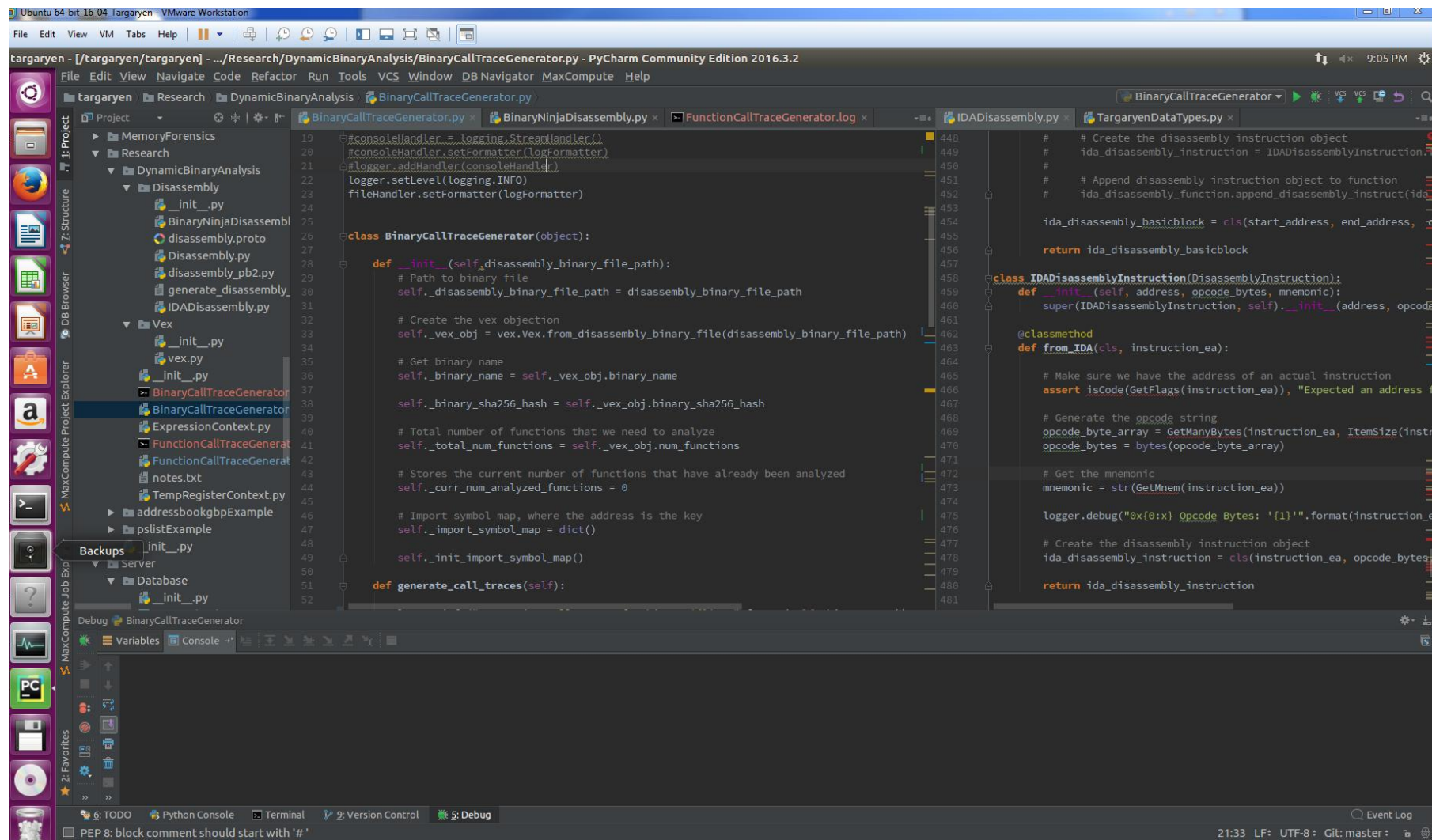


# (DEMO) CALL TRACE GENERATION

Target Binary: explorer.exe



# (DEMO) CALL TRACE GENERATION



```
19 #consoleHandler = logging.StreamHandler()
20 #consoleHandler.setFormatter(logFormatter)
21 #logger.addHandler(consoleHandler)
22 logger.setLevel(logging.INFO)
23 fileHandler.setFormatter(logFormatter)
24
25 class BinaryCallTraceGenerator(object):
26
27     def __init__(self, disassembly_binary_file_path):
28         # Path to binary file
29         self._disassembly_binary_file_path = disassembly_binary_file_path
30
31         # Create the vex object
32         self._vex_obj = vex.Vex.from_disassembly_binary_file(disassembly_binary_file_path)
33
34         # Get binary name
35         self._binary_name = self._vex_obj.binary_name
36
37         self._binary_sha256_hash = self._vex_obj.binary_sha256_hash
38
39         # Total number of functions that we need to analyze
40         self._total_num_functions = self._vex_obj.num_functions
41
42         # Stores the current number of functions that have already been analyzed
43         self._curr_num_analyzed_functions = 0
44
45         # Import symbol map, where the address is the key
46         self._import_symbol_map = dict()
47
48         self._init_import_symbol_map()
49
50     def generate_call_traces(self):
51
52         # Create the disassembly instruction object
53         ida_disassembly_instruction = IDADisassemblyInstruction(
54             # Append disassembly instruction object to function
55             # ida_disassembly_function.append_disassembly_instruction(ida_disassembly_instruction)
56
57             ida_disassembly_basicblock = cls(start_address, end_address,
58
59             return ida_disassembly_basicblock
60
61 class IDADisassemblyInstruction(DisassemblyInstruction):
62     def __init__(self, address, opcode_bytes, mnemonic):
63         super(IDADisassemblyInstruction, self).__init__(address, opcode_bytes, mnemonic)
64
65     @classmethod
66     def from_IDA(cls, instruction_ea):
67
68         # Make sure we have the address of an actual instruction
69         assert isCode(GetFlags(instruction_ea)), "Expected an address for instruction"
70
71         # Generate the opcode string
72         opcode_byte_array = GetManyBytes(instruction_ea, ItemSize(instruction_ea))
73         opcode_bytes = bytes(opcode_byte_array)
74
75         # Get the mnemonic
76         mnemonic = str(GetMnem(instruction_ea))
77
78         logger.debug("0x{:x} Opcode Bytes: '{1}'".format(instruction_ea, opcode_bytes))
79
80         # Create the disassembly instruction object
81         ida_disassembly_instruction = cls(instruction_ea, opcode_bytes, mnemonic)
82
83         return ida_disassembly_instruction
```



# (DEMO) CALL TRACE GENERATION

The screenshot displays the PyCharm IDE interface with the following components:

- File Explorer:** Shows the project structure with folders like 'Research' and 'DynamicBinaryAnalysis', and files like 'BinaryCallTraceGenerator.py', 'BinaryNinjaDisassembly.py', and 'disassembly.py'.
- Code Editor:** Displays the 'BinaryCallTraceGenerator.py' file. The code includes a search bar, a list of symbols, and a function 'generate\_call\_traces(self)' that logs call traces for a binary named 'explorer.exe'.
- Terminal:** Shows the output of the program, including the following log messages:

```
INFO:BinaryCallTraceGenerator:*****Test Harness*****
INFO:Disassembly.Disassembly:Serialized disassembly size: '4102 kb'
INFO:BinaryCallTraceGenerator:Generating Call Traces for binary 'explorer.exe'...
INFO:BinaryCallTraceGenerator:[0/3186] Generated Call Traces for function: 'sub_1000011E0'
INFO:FunctionCallGenerator:=====Call Trace =====
INFO:FunctionCallGenerator: 1) [0x1000011E0] Name:'sub_1000011E0' is_import='False' library_name: 'None' call_address: 0x1000011E0
INFO:FunctionCallGenerator: 2) [0x100040E74] Name:'sub_100040E74' is_import='False' library_name: 'None' call_address: 0x100040E74
INFO:FunctionCallGenerator:=====END Call Trace =====
INFO:FunctionCallGenerator:=====Call Trace =====
INFO:FunctionCallGenerator: 1) [0x1000011E0] Name:'sub_1000011E0' is_import='False' library_name: 'None' call_address: 0x1000011E0
INFO:FunctionCallGenerator:=====END Call Trace =====
INFO:FunctionCallGenerator:=====Call Trace =====
INFO:FunctionCallGenerator: 1) [0x1000011E0] Name:'sub_1000011E0' is_import='False' library_name: 'None' call_address: 0x1000011E0
INFO:FunctionCallGenerator:=====END Call Trace =====
```
- Bottom Bar:** Shows the status bar with the time '48:24', the file encoding 'UTF-8', the Git status 'Git: master', and the PyCharm version 'PyCharm Community Edition is ready to update. (11/3/17 3:18 AM)'.



# (DEMO) CALL TRACE GENERATION

```
INFO:FunctionCallGenerator: 19) [0x100093652] Name:'GetWindowLongW' is_import='True' library_name: 'USER32' call_address: 0x1000bae
INFO:FunctionCallGenerator: 20) [0x100093672] Name:'SelectObject' is_import='True' library_name: 'GDI32' call_address: 0x1000ba748
INFO:FunctionCallGenerator:=====END Call Trace =====
```

```
INFO:FunctionCallGenerator:=====Call Trace =====
```

```
INFO:FunctionCallGenerator: 1) [0x1000011e0] Name:'sub_1000011E0' is_import='False' library_name: 'None' call_address: 0x1000011e0
INFO:FunctionCallGenerator: 2) [0x100040ea9] Name:'sub_100040E74' is_import='False' library_name: 'None' call_address: 0x100040e74
INFO:FunctionCallGenerator: 3) [0x100046324] Name:'SHIsChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x1000bb
INFO:FunctionCallGenerator: 4) [0x100046355] Name:'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38
INFO:FunctionCallGenerator: 5) [0x10006670a] Name:'QueryServiceExec' is_import='True' library_name: 'SHLWAPI' call_address
INFO:FunctionCallGenerator: 6) [0x100047814] Name:'sub_100047814' is_import='False' library_name: 'None' call_address: 0x100047814
INFO:FunctionCallGenerator: 7) [0x1000ba968] Name:'sub_1000ba968' is_import='True' library_name: 'USER32' call_address: 0x1000ba968
INFO:FunctionCallGenerator: 8) [0x100050b80] Name:'sub_100050b80' is_import='False' library_name: 'None' call_address: 0x100050b80
INFO:FunctionCallGenerator: 9) [0x10005087c] Name:'sub_10005087c' is_import='False' library_name: 'None' call_address: 0x10005087c
INFO:FunctionCallGenerator: 10) [0x100050c33] Name:'sub_100050c33' is_import='False' library_name: 'None' call_address: 0x100050c33
INFO:FunctionCallGenerator: 11) [0x100050c44] Name:'sub_100047198' is_import='False' library_name: 'None' call_address: 0x100047198
INFO:FunctionCallGenerator: 12) [0x100050c7d] Name:'GetDC' is_import='True' library_name: 'USER32' call_address: 0x1000ba9c0
INFO:FunctionCallGenerator: 13) [0x100050cce] Name:'GetWindowLongW' is_import='True' library_name: 'USER32' call_address: 0x1000bae
INFO:FunctionCallGenerator: 14) [0x100050d21] Name:'sub_100050D68' is_import='False' library_name: 'None' call_address: 0x100050d68
INFO:FunctionCallGenerator: 15) [0x100050dc5] Name:'memset' is_import='False' library_name: 'None' call_address: 0x100001318
INFO:FunctionCallGenerator: 16) [0x10006aa36] Name:'sub_100093600' is_import='False' library_name: 'None' call_address: 0x100093600
INFO:FunctionCallGenerator: 17) [0x10009361f] Name:'GetDC' is_import='True' library_name: 'USER32' call_address: 0x1000ba9c0
INFO:FunctionCallGenerator: 18) [0x100093634] Name:'CreateCompatibleDC' is_import='True' library_name: 'GDI32' call_address: 0x1000
INFO:FunctionCallGenerator: 19) [0x100093652] Name:'GetWindowLongW' is_import='True' library_name: 'USER32' call_address: 0x1000bae
INFO:FunctionCallGenerator: 20) [0x100093672] Name:'SelectObject' is_import='True' library_name: 'GDI32' call_address: 0x1000ba748
INFO:FunctionCallGenerator: 21) [0x1000936ed] Name:'BitBlt' is_import='True' library_name: 'GDI32' call_address: 0x1000ba718
INFO:FunctionCallGenerator: 22) [0x100093732] Name:'StretchBlt' is_import='True' library_name: 'GDI32' call_address: 0x1000ba650
INFO:FunctionCallGenerator:=====END Call Trace =====
```

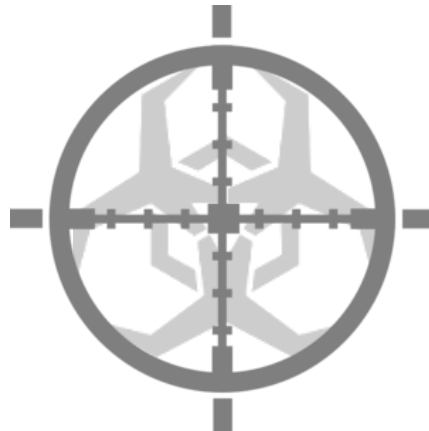
```
INFO:FunctionCallGenerator:=====Call Trace =====
```

```
INFO:FunctionCallGenerator: 1) [0x1000011e0] Name:'sub_1000011E0' is_import='False' library_name: 'None' call_address: 0x1000011e0
INFO:FunctionCallGenerator: 2) [0x100046324] Name:'SHIsChildOrSelf' is_import='True' library_name: 'SHLWAPI' call_address: 0x1000bb
INFO:FunctionCallGenerator: 3) [0x100046355] Name:'GetParent' is_import='True' library_name: 'USER32' call_address: 0x1000bab38
```



Leverage Call Trace Info to Detect APTs

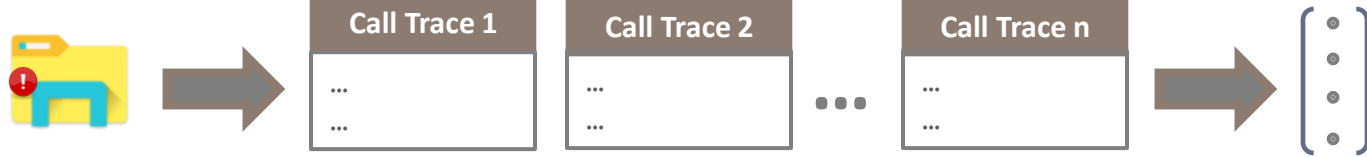
## PHASE II DETECTION





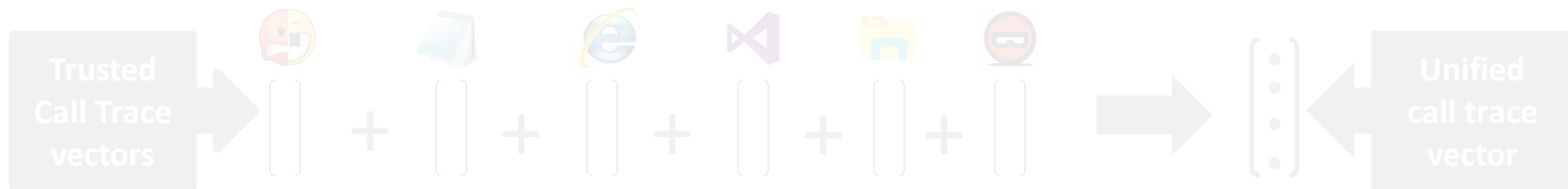
# LEVERAGE CALL TRACE INFO TO DETECT APTs

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

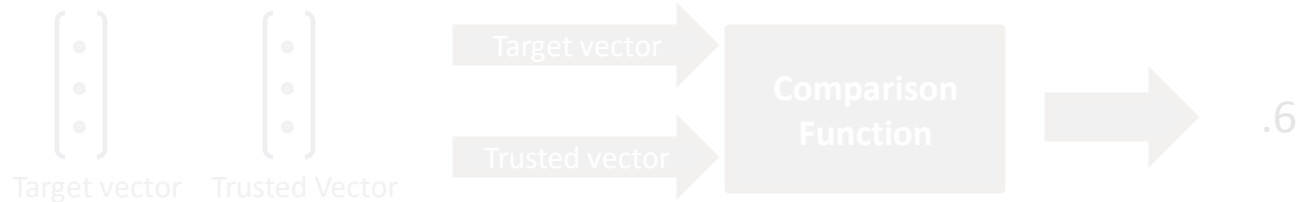


Traces generated via **dynamic analysis**, where different code paths are traversed to maximize code coverage

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



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



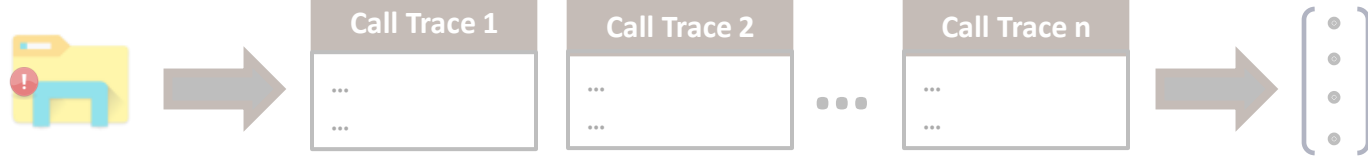
Function output is percentage of call sequences in target that are trusted (1 → all sequences trusted)





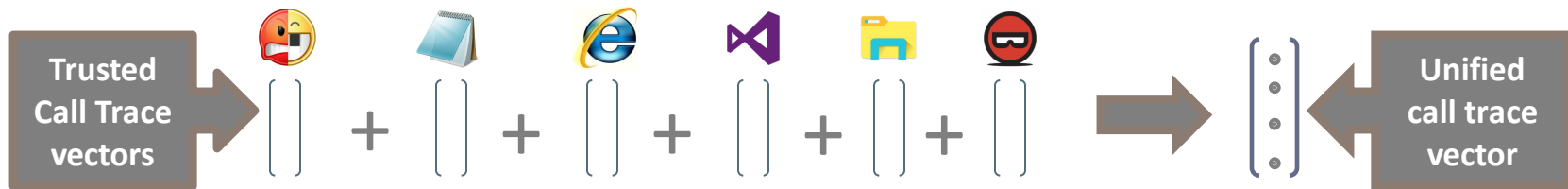
# LEVERAGE CALL TRACE INFO TO DETECT APTs

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

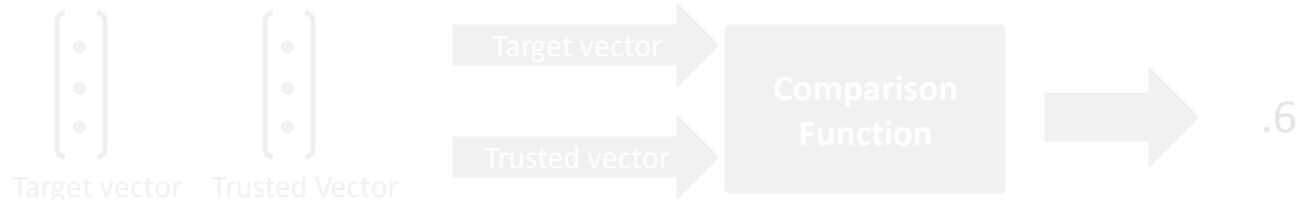


*Traces generated via **dynamic analysis**, where different code paths are traversed to maximize code coverage*

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



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

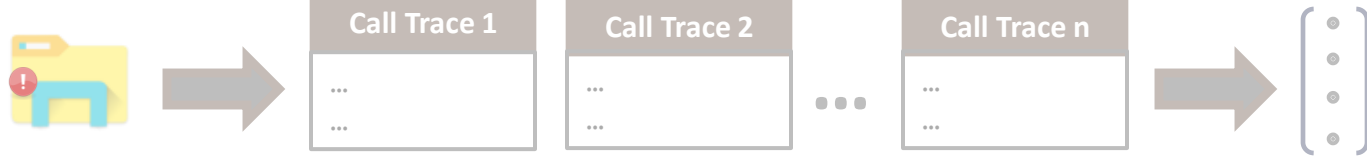


*Function output is percentage of call sequences in target that are trusted (1 → all sequences trusted)*



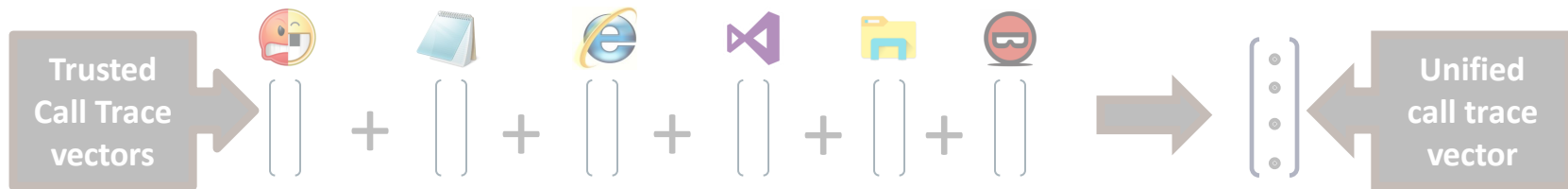
# LEVERAGE CALL TRACE INFO TO DETECT APTs

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

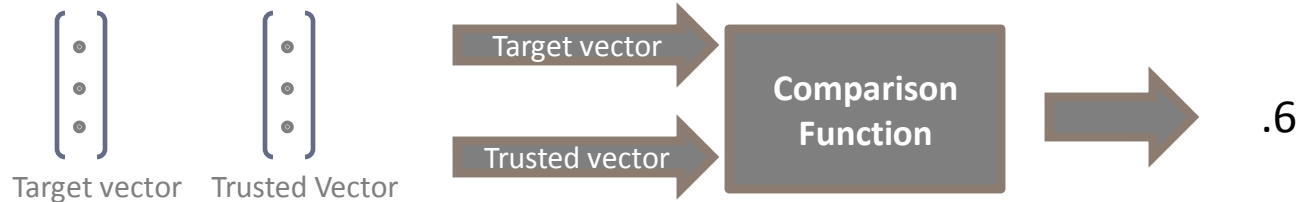


*Traces generated via **dynamic analysis**, where different code paths are traversed to maximize code coverage*

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



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

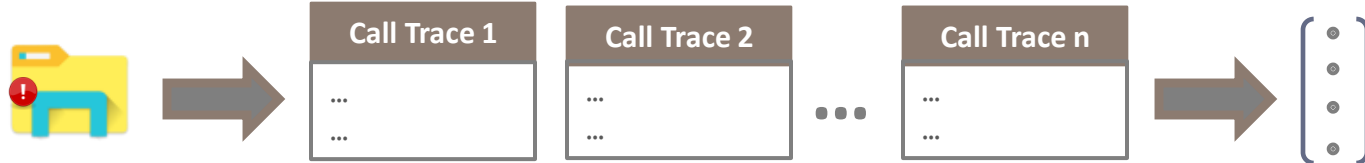


*Function output is percentage of call sequences in target that are trusted (1 → all sequences trusted)*



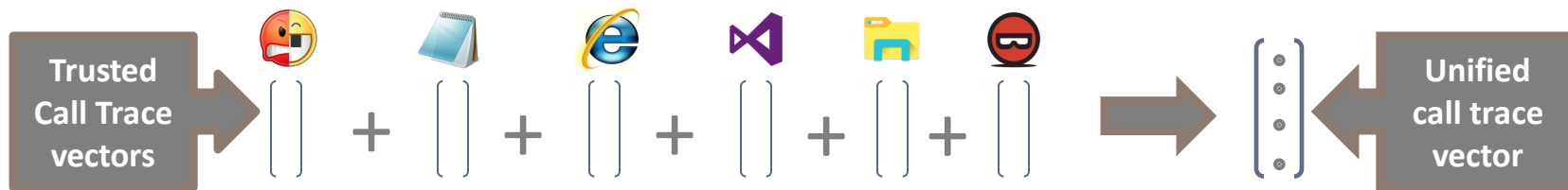
# LEVERAGE CALL TRACE INFO TO DETECT APTs

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

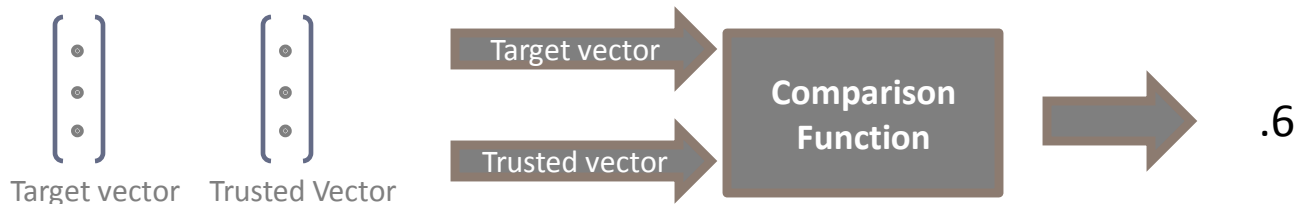


*Traces generated via **dynamic analysis**, where different code paths are traversed to maximize code coverage*

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



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

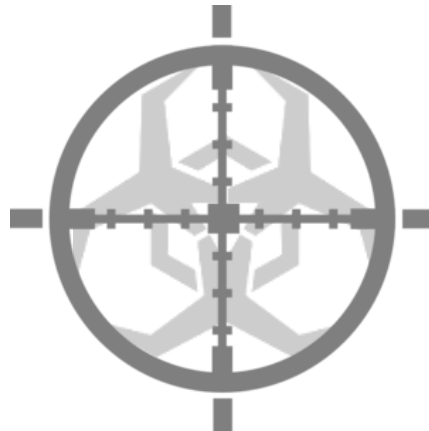


*Function output is percentage of call sequences in target that are trusted (1 → all sequences trusted)*

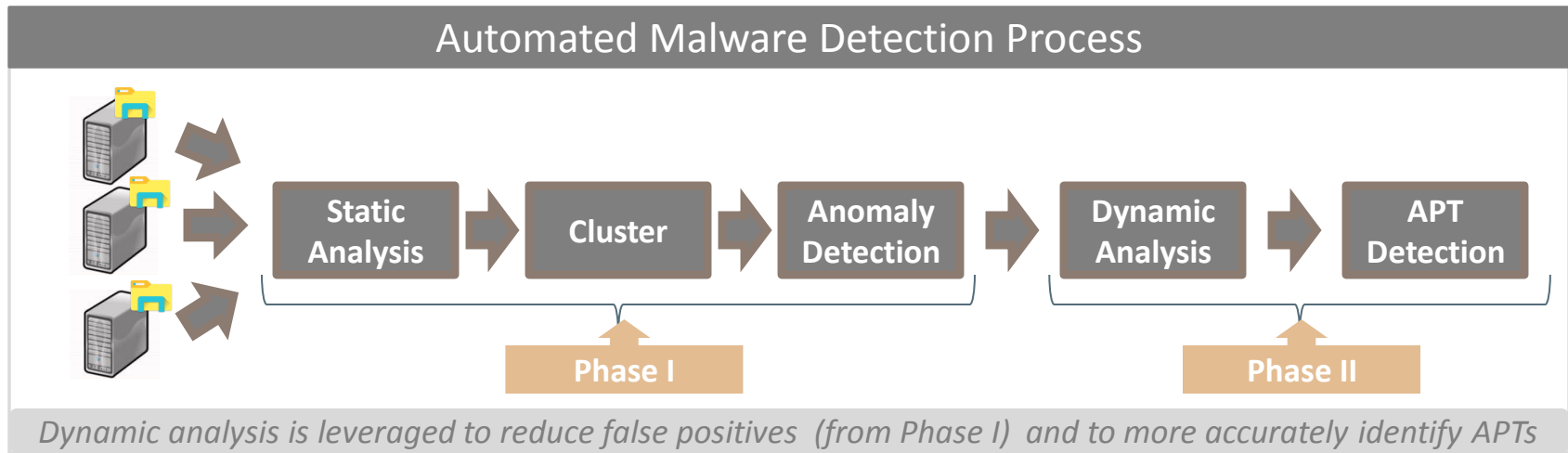
(See Appendix E for more details about the algorithm)



# CONCLUSION



# 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*)



# REFERENCES

1. Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 639-668.
2. Stuttgen, Johannes & Cohen, Michael (2013). Anti-forensic Resilient Memory Acquisition. *Digit. Investig.*, 10, S105-S115..
3. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., & Vigna, G. (2015, February). Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*.
4. Koret, Joxean & Bachaalany, Elias (2015). *The Antivirus Hacker's Handbook*. Wiley Publishing
5. Carter, K. M., Lippmann, R. P., & Boyer, S. W. (2010, November). Temporally oblivious anomaly detection on large networks using functional peers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement* (pp. 465-471). ACM.
6. Mosli, R., Li, R., Yuan, B., & Pan, Y. (2016, May). Automated malware detection using artifacts in forensic memory images. In *Technologies for Homeland Security (HST), 2016 IEEE Symposium on* (pp. 1-6). IEEE.



# REFERENCES

6. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., ... & Vigna, G. (2016, February). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS* (Vol. 16, pp. 1-16).
7. Liang, S. C. (2016). Understanding behavioural detection of antivirus.
8. Anderson, B., Storlie, C., & Lane, T. (2012, October). Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence* (pp. 3-14). ACM. Koret, Joxean & Bachaalany, Elias (2015).
9. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., ... & Vigna, G. (2016, May). Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on* (pp. 138-157). IEEE.
10. Ravi, C., & Manoharan, R. (2012). Malware detection using windows api sequence and machine learning. *International Journal of Computer Applications*, 43(17), 12-16.





# Q&A

- 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



# Q&A

- 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.



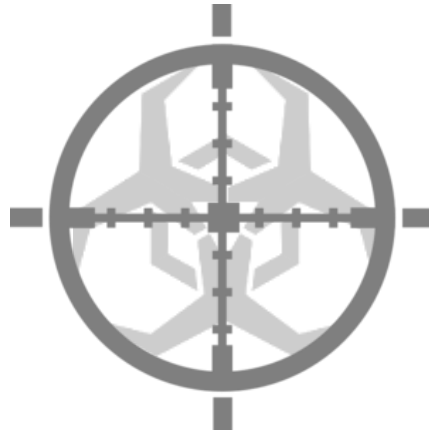
# APPENDIX

- Machine Learning Primer A
- Advanced Binary Analysis B
- Memory Forensics C
- Binary Vector Generation D
- Call Trace Vector Generation E
- Hooking F



# Machine Learning Primer

## APPENDIX A



# MACHINE LEARNING

## ■ 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

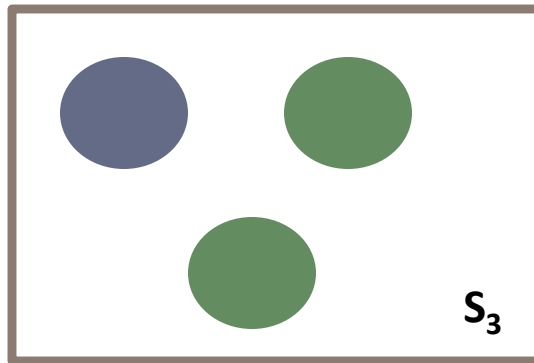
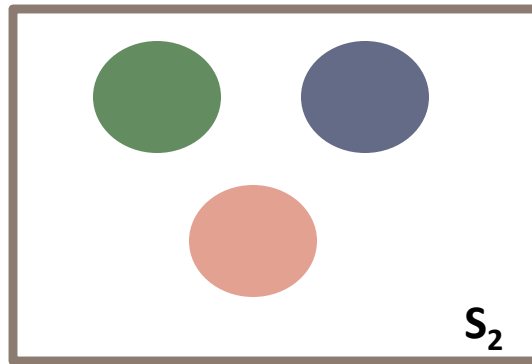
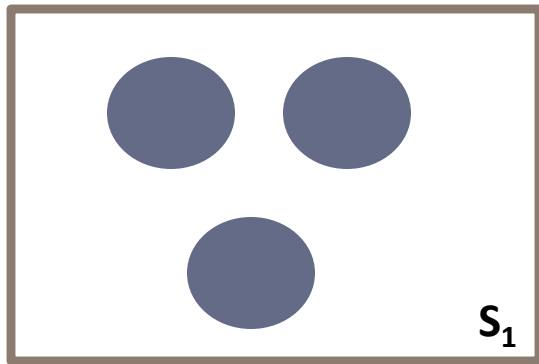
## ■ Challenges/Steps

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)



# MACHINE LEARNING

- **Example:** Group squares based on similarity of the color types



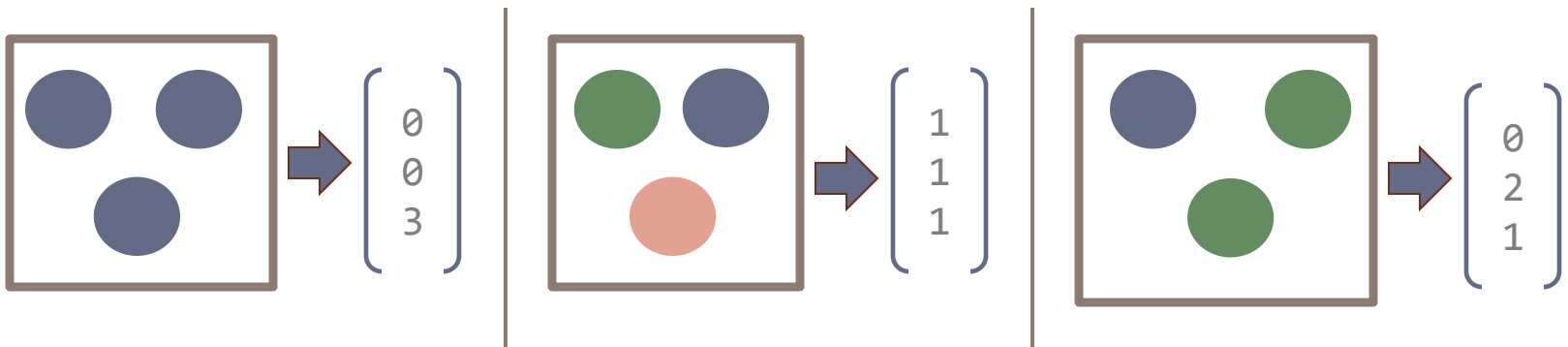
# MACHINE LEARNING

## 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

$\begin{pmatrix} \text{Pink} \\ \text{Green} \\ \text{Blue} \end{pmatrix}$

- Vector representations





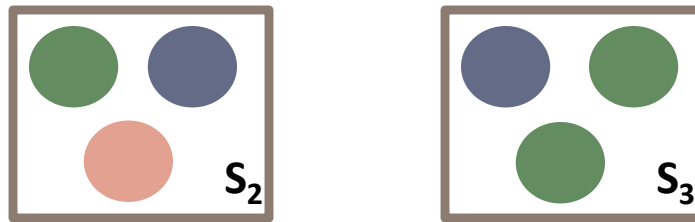
# MACHINE LEARNING

## 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:



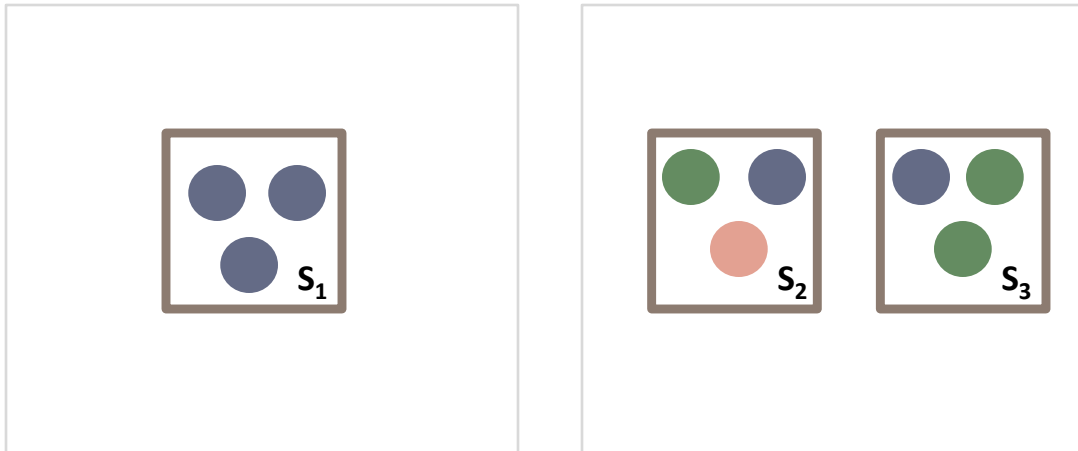
$$J(S_2, S_3) = \frac{S_2 \cap S_3}{S_2 \cup S_3} = \frac{2}{3}$$



# MACHINE LEARNING

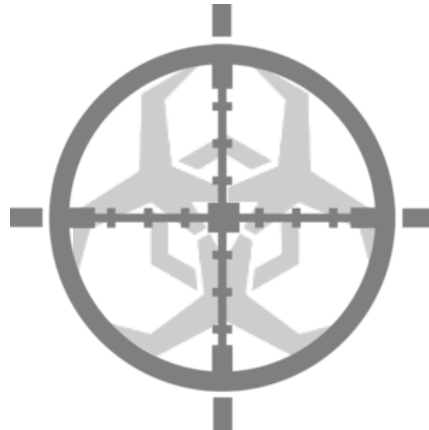
## 3. Selecting a suitable algorithm

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



Binary Analysis

# APPENDIX B



# 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)



# BINARY ANALYSIS

## ■ Motivational example for Symbolic Execution

```
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();
    }
}
```



# BINARY ANALYSIS

## ■ Motivational example for Symbolic Execution

```
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





# BINARY ANALYSIS

- Analyzing the call traces of example

Call Trace A
main() doBenignStuff()

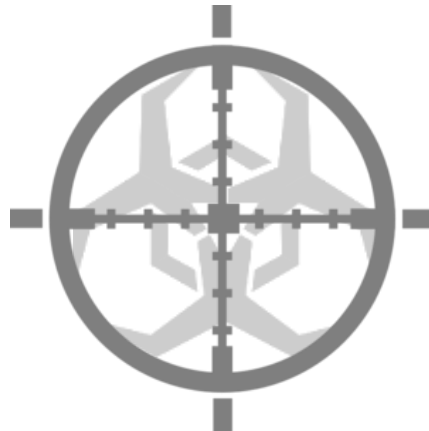
Call Trace B
main() doReallyBadStuff()

- **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

# APPENDIX C



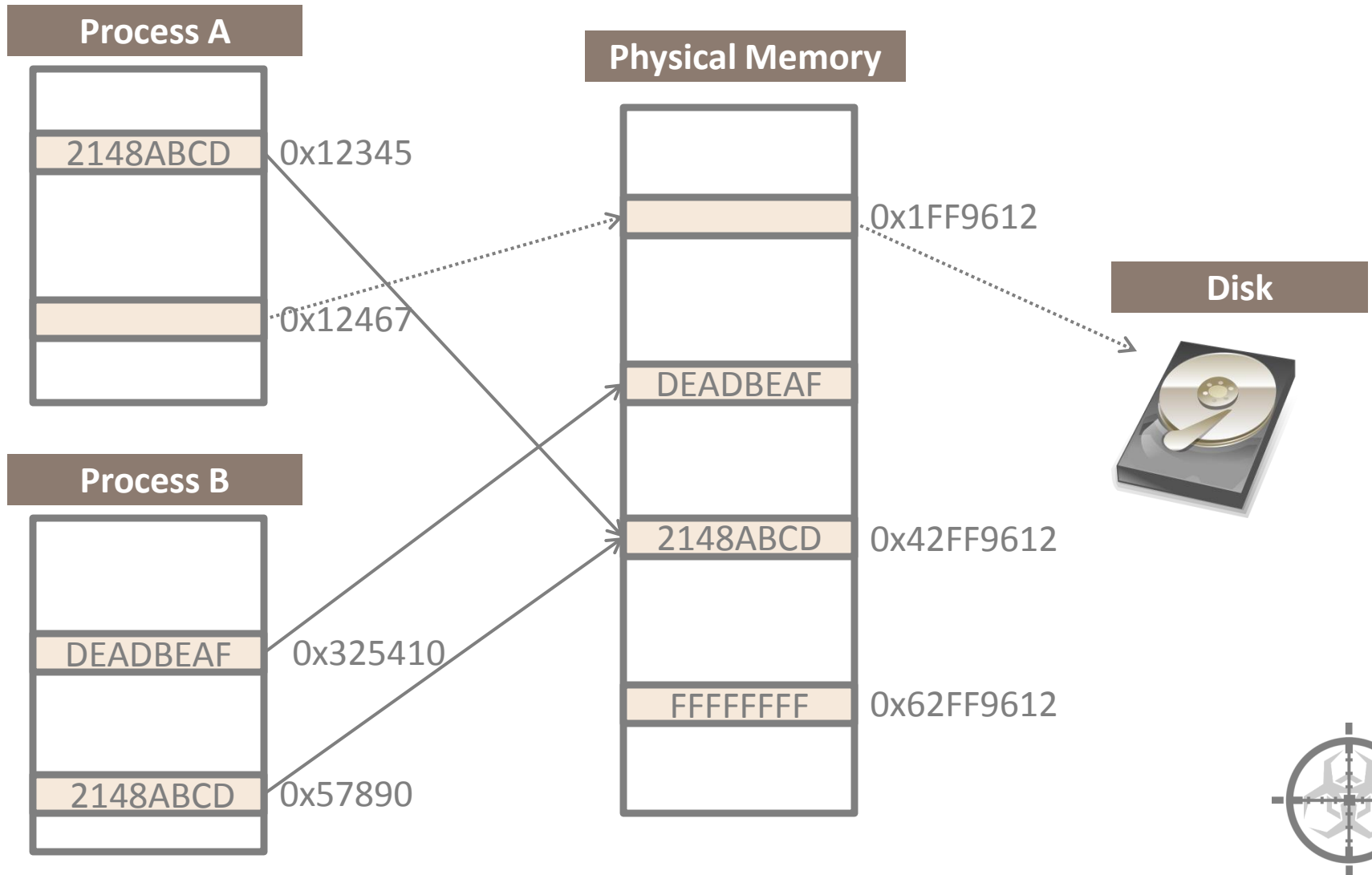
# MEMORY FORENSICS

- **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



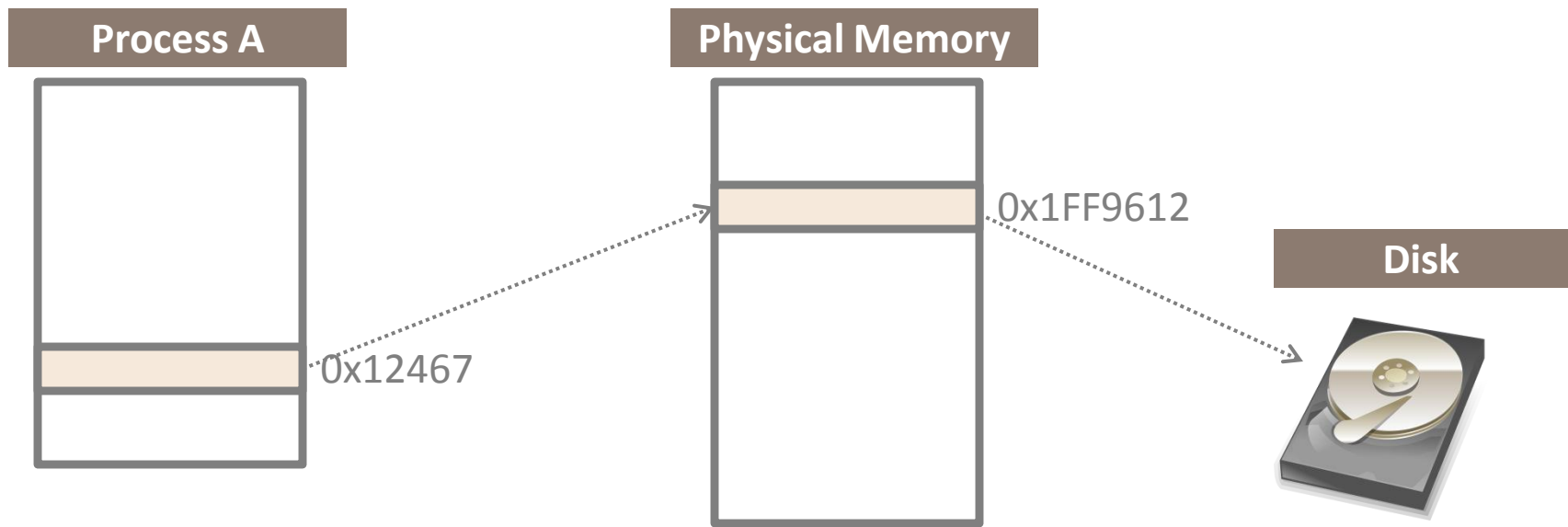
# MEMORY FORENSICS

- Virtual Addressing and Memory Acquisition



# MEMORY FORENSICS

- 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***



# MEMORY FORENSICS

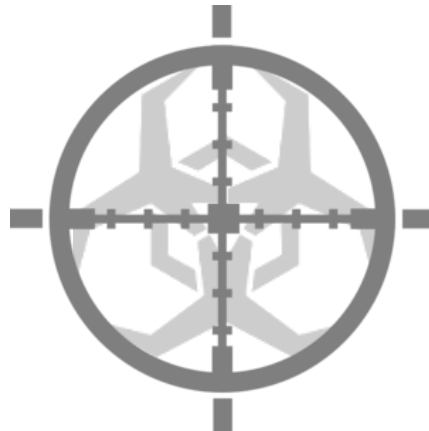
## ■ Anti-Forensics

- 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*

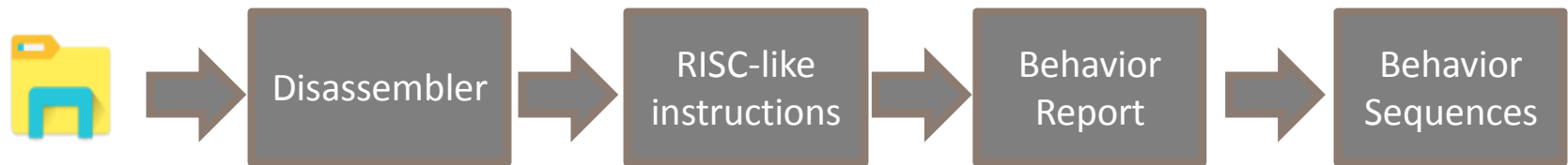


# Generating a Vector Representation of a Binary

## APPENDIX D



# BINARY VECTOR GENERATION



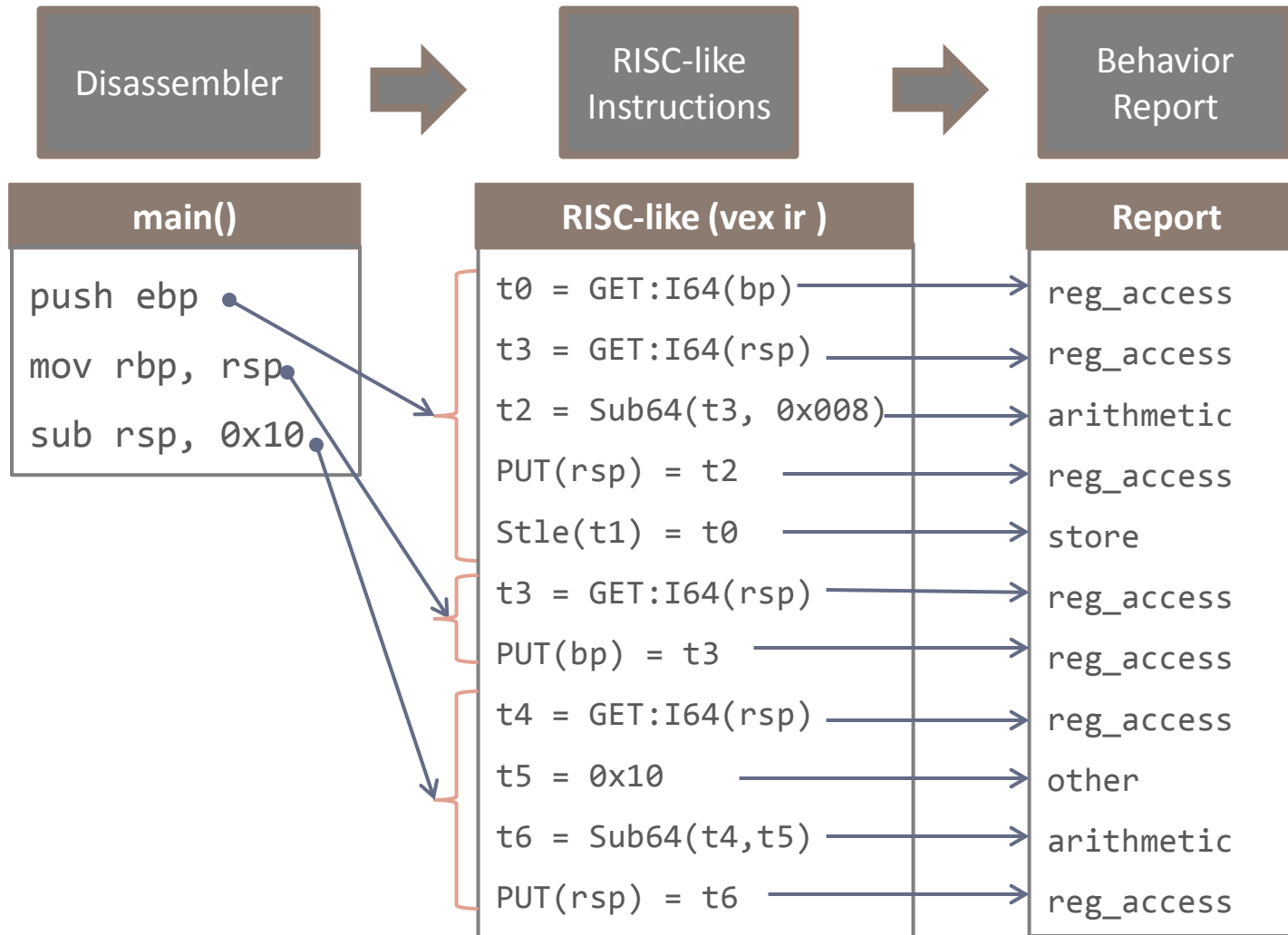
## ■ Binary Vector Generation

- 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

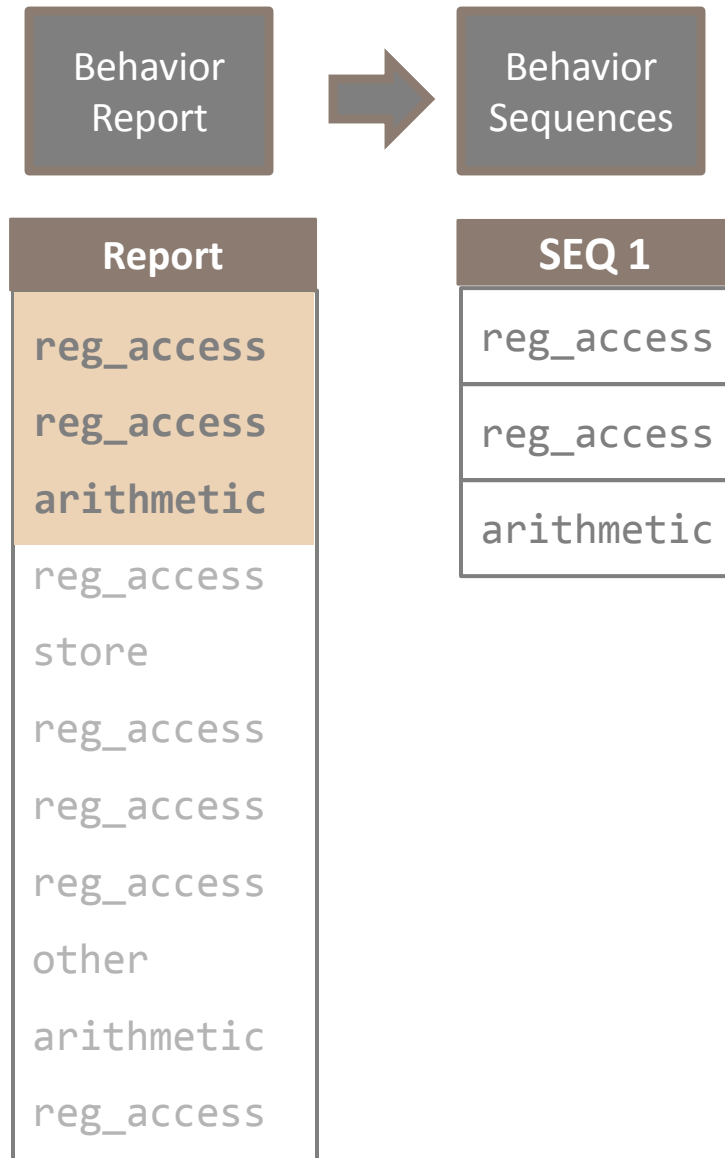




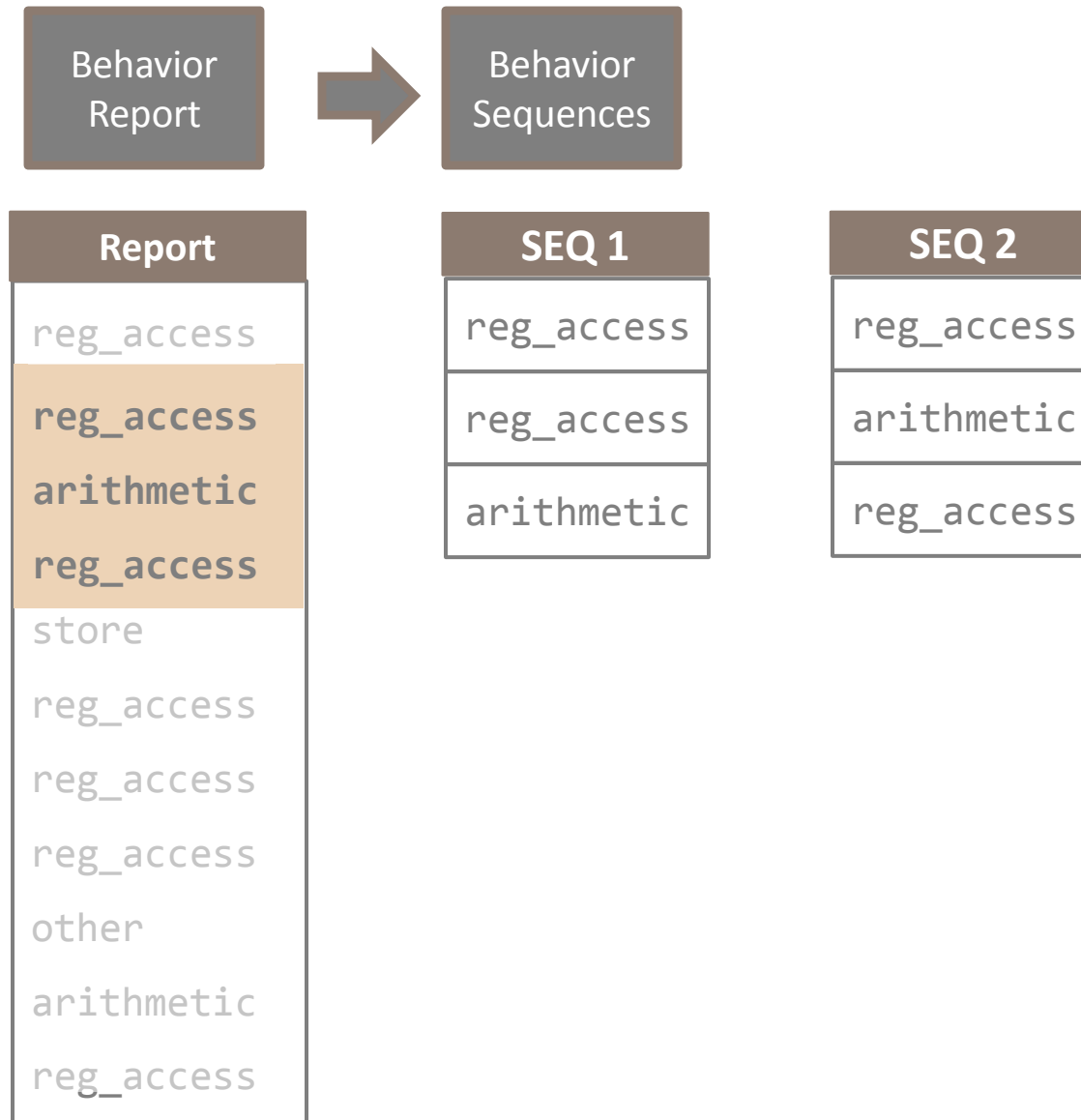
# BINARY VECTOR GENERATION



# BINARY VECTOR GENERATION



# BINARY VECTOR GENERATION



# BINARY VECTOR GENERATION

Behavior  
Report



Behavior  
Sequences

Report

reg\_access

reg\_access

**arithmetic**

**reg\_access**

**store**

reg\_access

reg\_access

reg\_access

other

arithmetic

reg\_access

SEQ 1

reg\_access

reg\_access

arithmetic

SEQ 2

reg\_access

arithmetic

reg\_access

SEQ 3

arithmetic

reg\_access

store



# BINARY VECTOR GENERATION

Behavior  
Report



Behavior  
Sequences

**Report**

reg\_access  
reg\_access  
arithmetic  
**reg\_access**  
**store**  
**reg\_access**  
reg\_access  
reg\_access  
other  
arithmetic  
reg\_access

**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



Behavior  
Sequences

Report

reg\_access  
reg\_access  
arithmetic  
reg\_access  
**store**  
**reg\_access**  
**reg\_access**  
reg\_access  
other  
arithmetic  
reg\_access

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

SEQ 5

store  
reg\_access  
reg\_access



# BINARY VECTOR GENERATION

Behavior  
Report



Behavior  
Sequences

Report

reg\_access  
reg\_access  
arithmetic  
reg\_access  
store

reg\_access  
reg\_access  
reg\_access

other  
arithmetic  
reg\_access

SEQ 1

reg\_access  
reg\_access  
arithmetic

SEQ 2

reg\_access  
arithmetic  
reg\_access

SEQ 3

arithmetic  
reg\_access  
store

SEQ 4

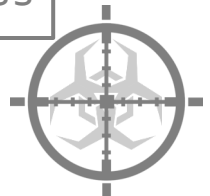
reg\_access  
store  
reg\_access

SEQ 5

store  
reg\_access  
reg\_access

SEQ 6

reg\_access  
reg\_access  
reg\_access



# BINARY VECTOR GENERATION

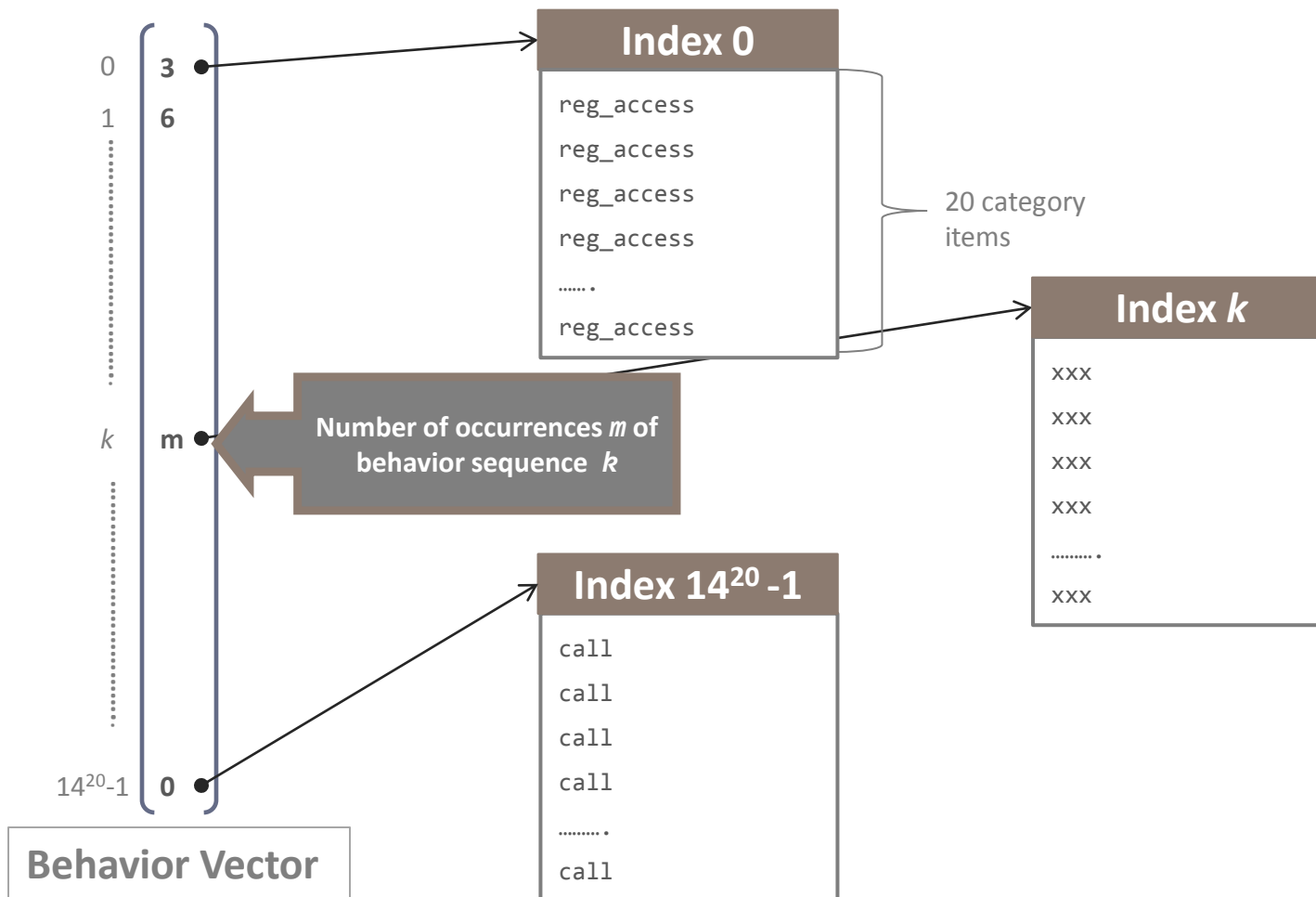
- Total possible behavior sequence permutations:  $\sim 2^{77}$ 
  - Number of instruction categories (e.g. store and branch): 14
  - Length of behavior sequence: 20
  - $14^{20} \sim 2^{77}$
- **Naive Approach**
  - Create a  $14^{20}$  dimensional vector to express binary behavior
  - ***Each vector dimension maps to a unique behavior sequence***
  - Number stored at dimension  $k$  is the number of times behavior sequence occurs in executable





# BINARY VECTOR GENERATION

## ■ Naive Approach (continued...)



# 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'

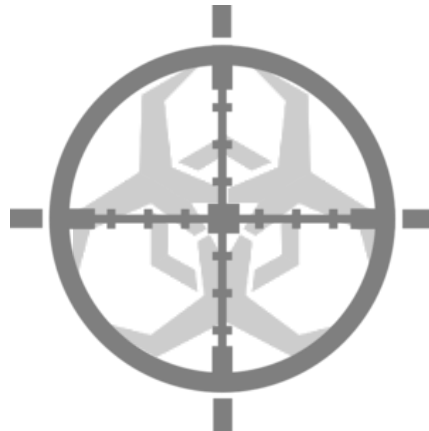
## ▪ **Better Approach**

- 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



# Efficient Diffing Algorithm

## APPENDIX D



# EFFICIENT DIFFING ALGOIRHTM

Step 2: Compute the similarity of a pair of vectors



*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:

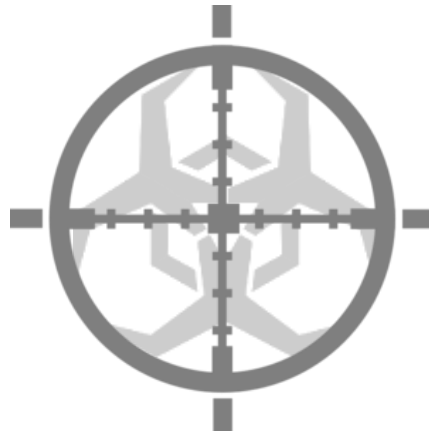
$$J(S_i, S_j) = \frac{S_i \cap S_j}{S_i \cup S_j} = \frac{\text{\#common behavior sequences}}{\text{\#total behaviors sequences}}$$

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



# Call Trace Vector Generation

## APPENDIX E



# 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



Host F

## 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()



Trace  
Sequences

## Call Trace (Code Path n)

main()

**LoadLibraryW()**

**RegisterShellHook()**

**GetTokenInformation()**

LsaOpenPolicy()

VirtualAlloc()

CreateEventW()

LogoffWindowsDialog()

## Trace Seq 1

main()

LoadLibraryW()

RegisterShellHook()

## Trace Seq 2

LoadLibraryW()

RegisterShellHook()

GetTokenInformation()



# 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()
```

## Trace Seq 3

```
RegisterShellHook()  
GetTokenInformation()  
LsaOpenPolicy()
```

## Trace Seq 2

```
LoadLibraryW()  
RegisterShellHook()  
GetTokenInformation()
```



# 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()
```

## Trace Seq 3

```
RegisterShellHook()  
GetTokenInformation()  
LsaOpenPolicy()
```

## Trace Seq 2

```
LoadLibraryW()  
RegisterShellHook()  
GetTokenInformation()
```

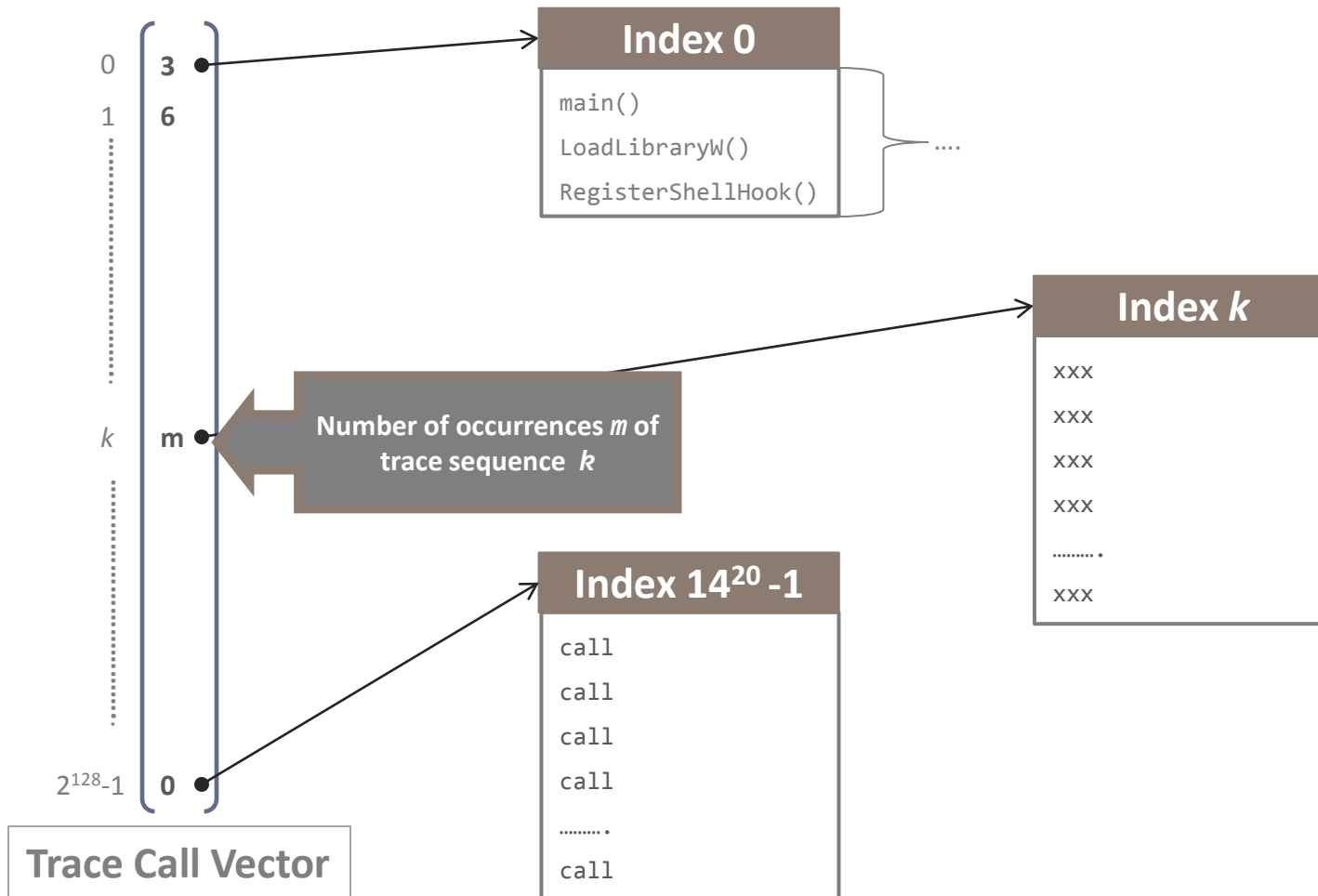
## Trace Seq 4

```
GetTokenInformation()  
LsaOpenPolicy()  
VirtualAlloc()
```

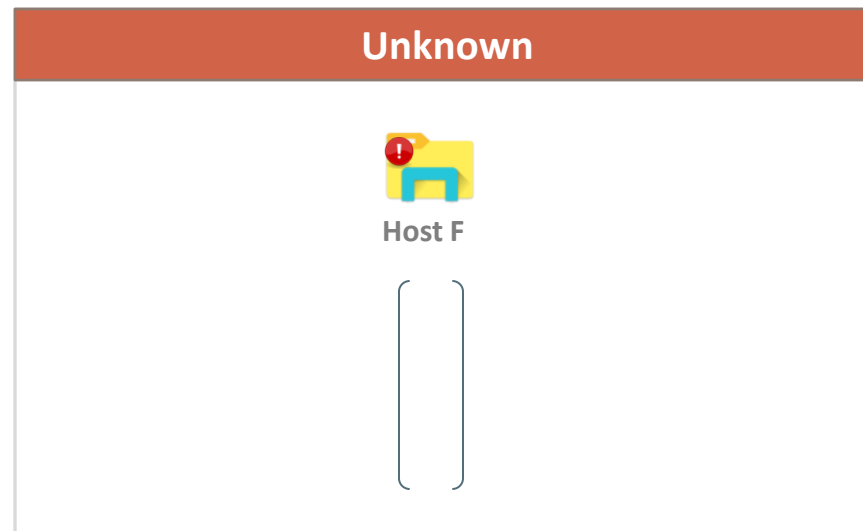
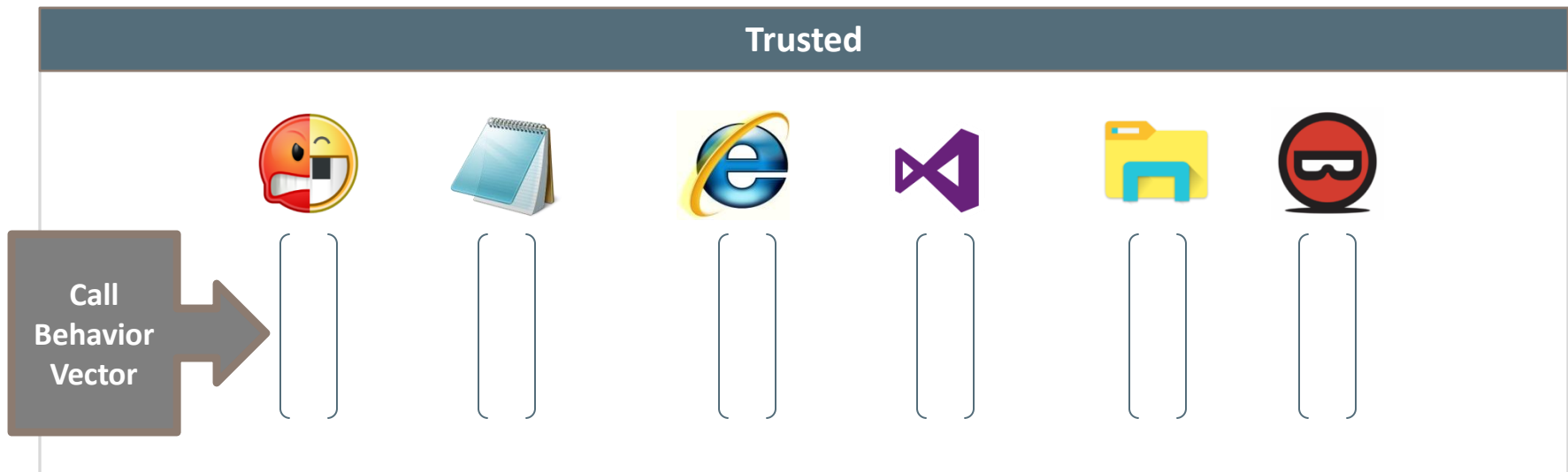


# CALL TRACE VECTOR GENERATION

## Trace Call Vector

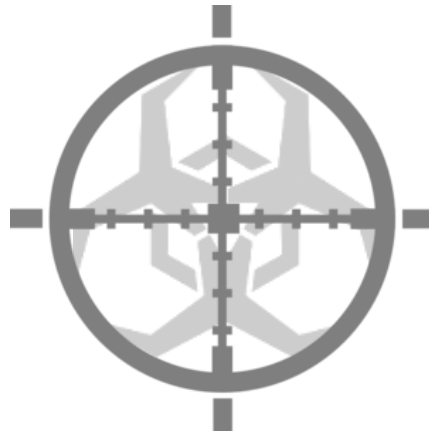


# CALL TRACE VECTOR GENERATION

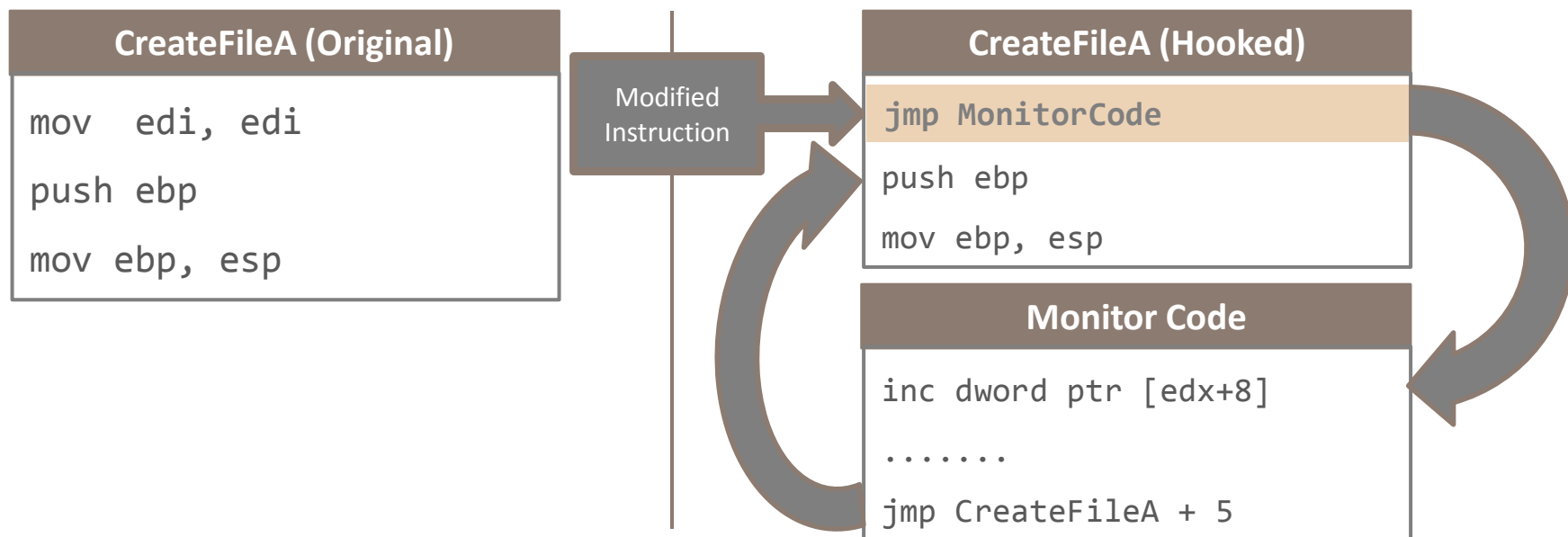


Hooking

# 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***

