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)

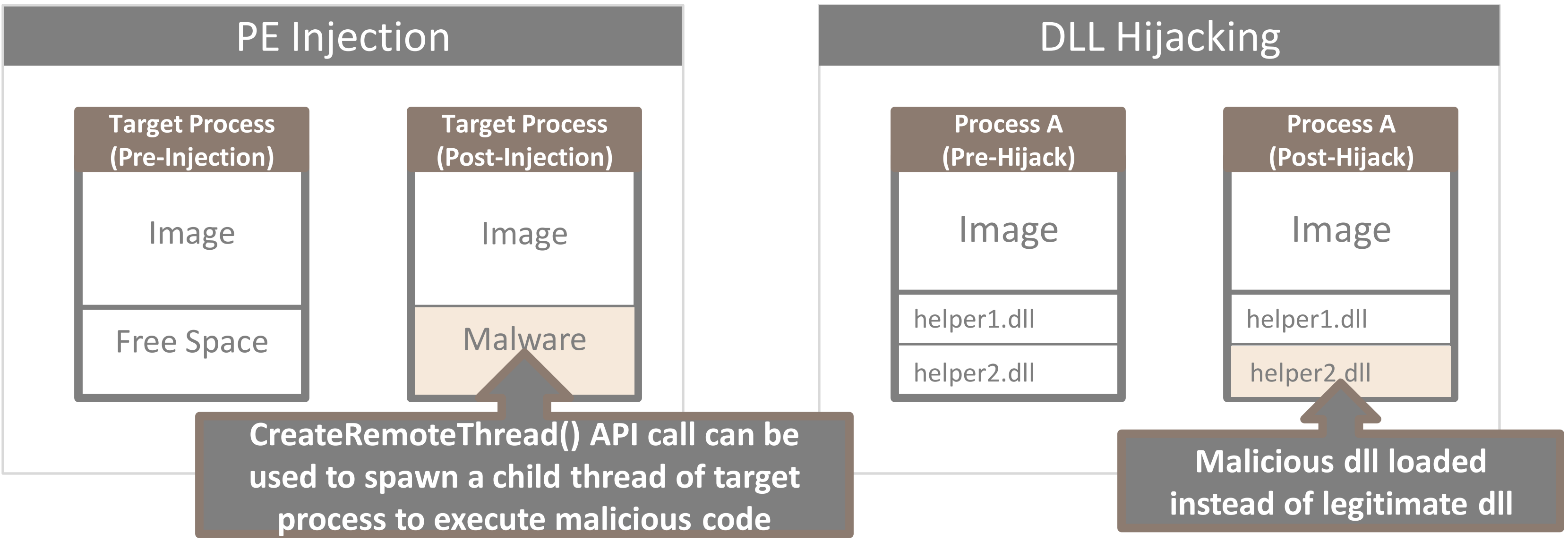
OUTLINE

- 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

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***

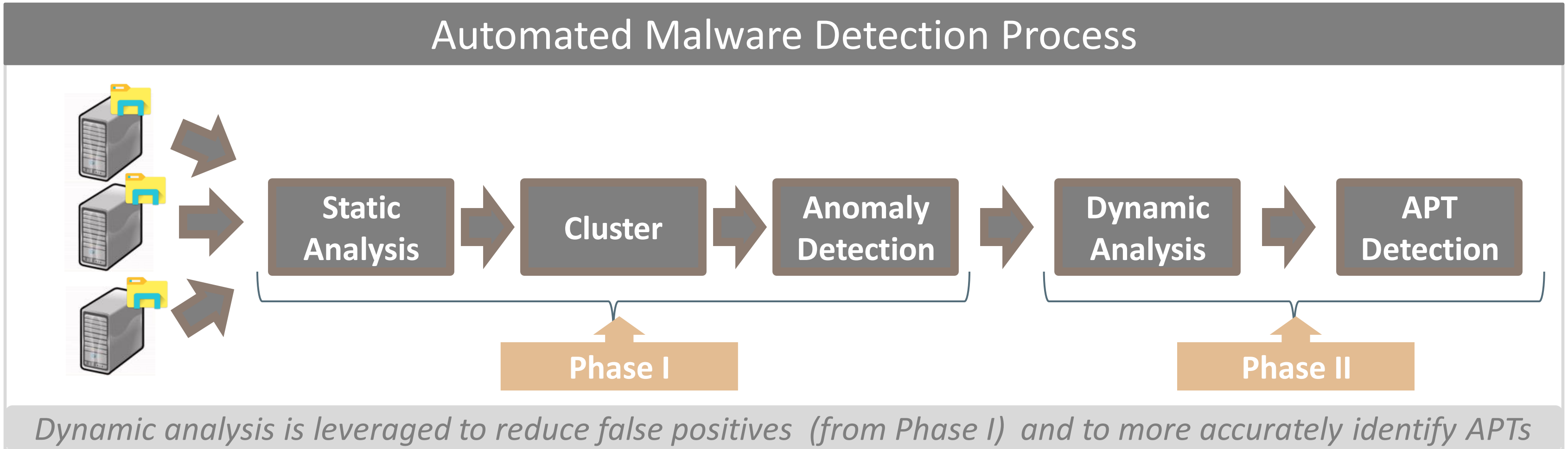
- *As a consequence, periodic live collection and analysis of memory artifacts can provide an effective means to identify malware residing on a host*

OBJECTIVE

- **Demonstrate** an approach to compare the following ***memory artifacts*** across a set of networked hosts ***in a scalable manner*** to identify anomalous code:
  1. Processes
  2. Shared Libraries
  3. Kernel Modules
  4. Drivers

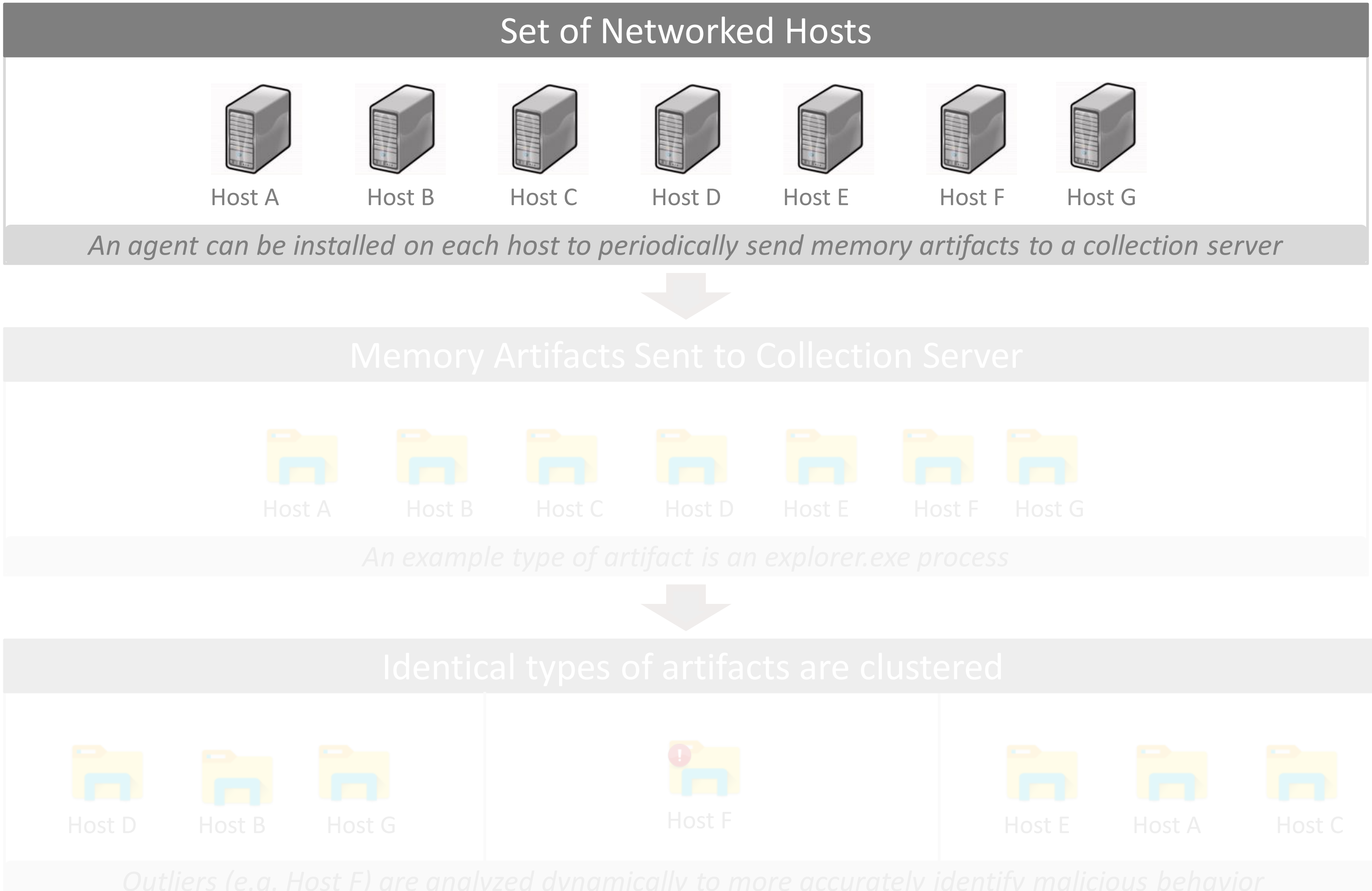
- Specifically, we will discuss how an ***approximate clustering algorithm*** with ***linear run-time performance*** can be leveraged to identify outliers among a set of equivalent types of artifacts (e.g. explorer.exe processes) collected from each networked host
- *We will also discuss how **Dynamic Binary Analysis** can be utilized to improve detection of sophisticated malware threats*

MAIN TAKEAWAYS

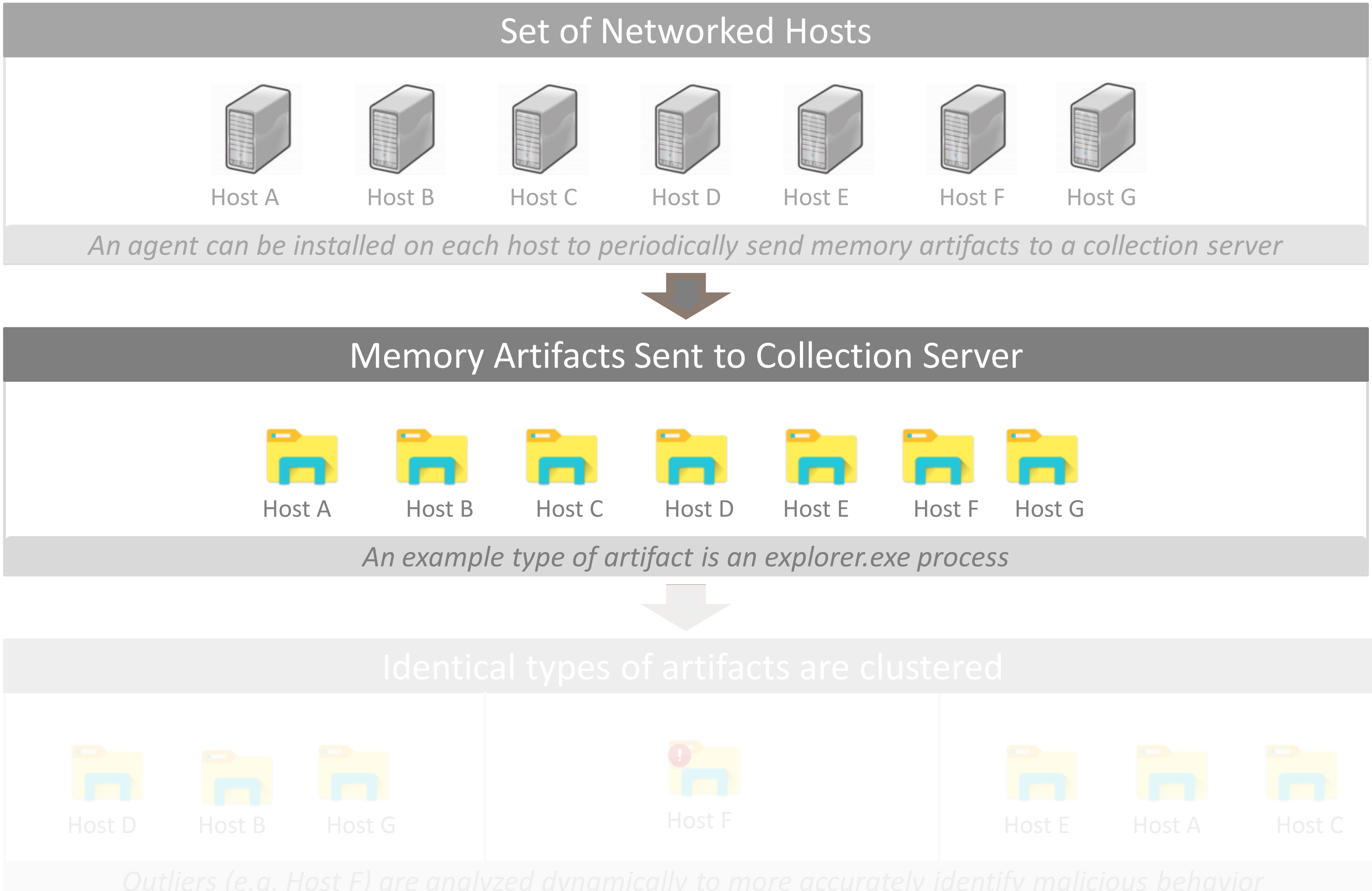


- **Phase I** : We'll leverage **static analysis** to provide us with a computationally efficient way to rapidly identify memory artifacts with anomalous code
- **Phase II**: Dynamic Analysis will be utilized to differentiate between **benign anomalous code** and **malware**

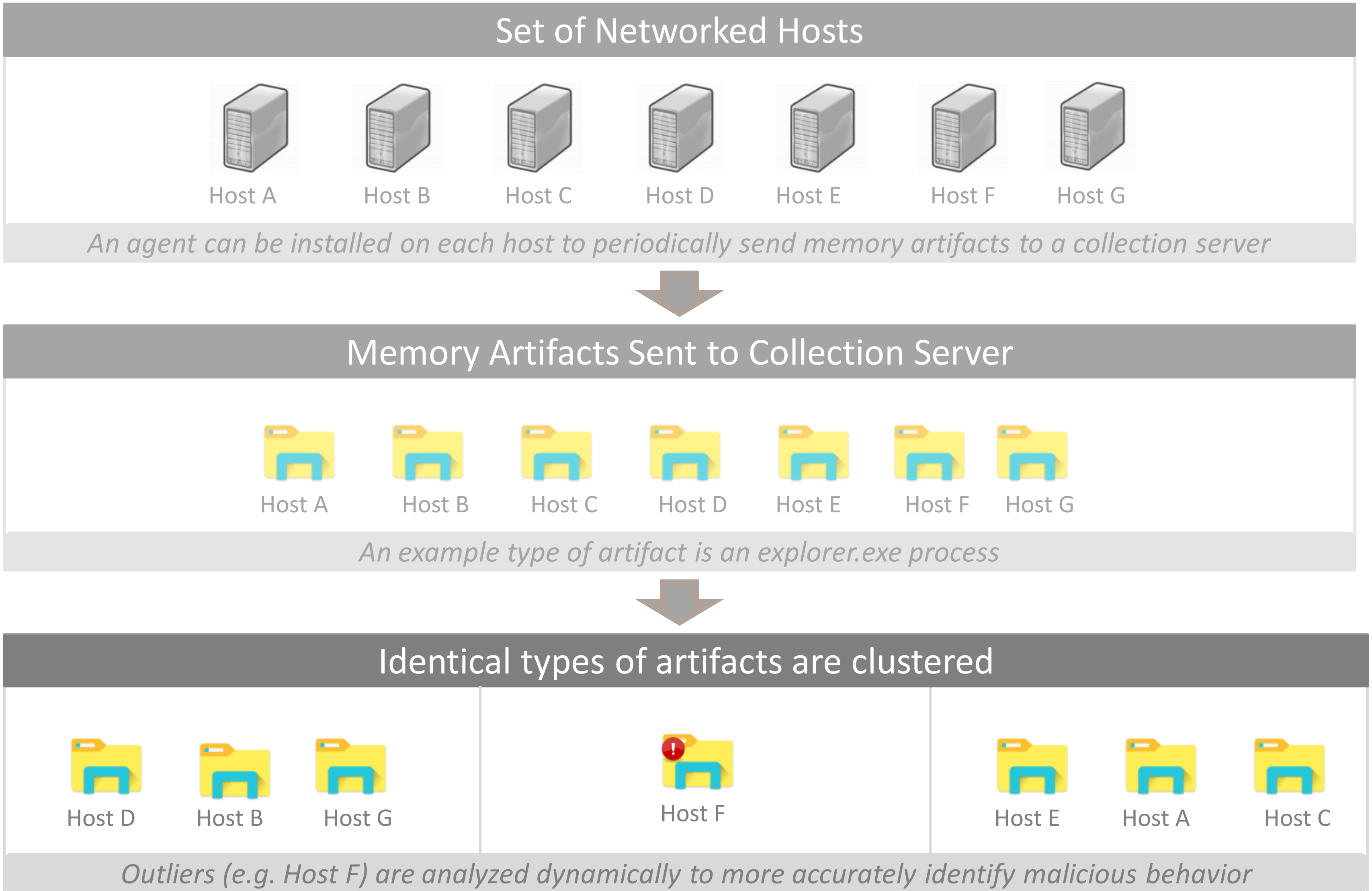
Main Takeaways



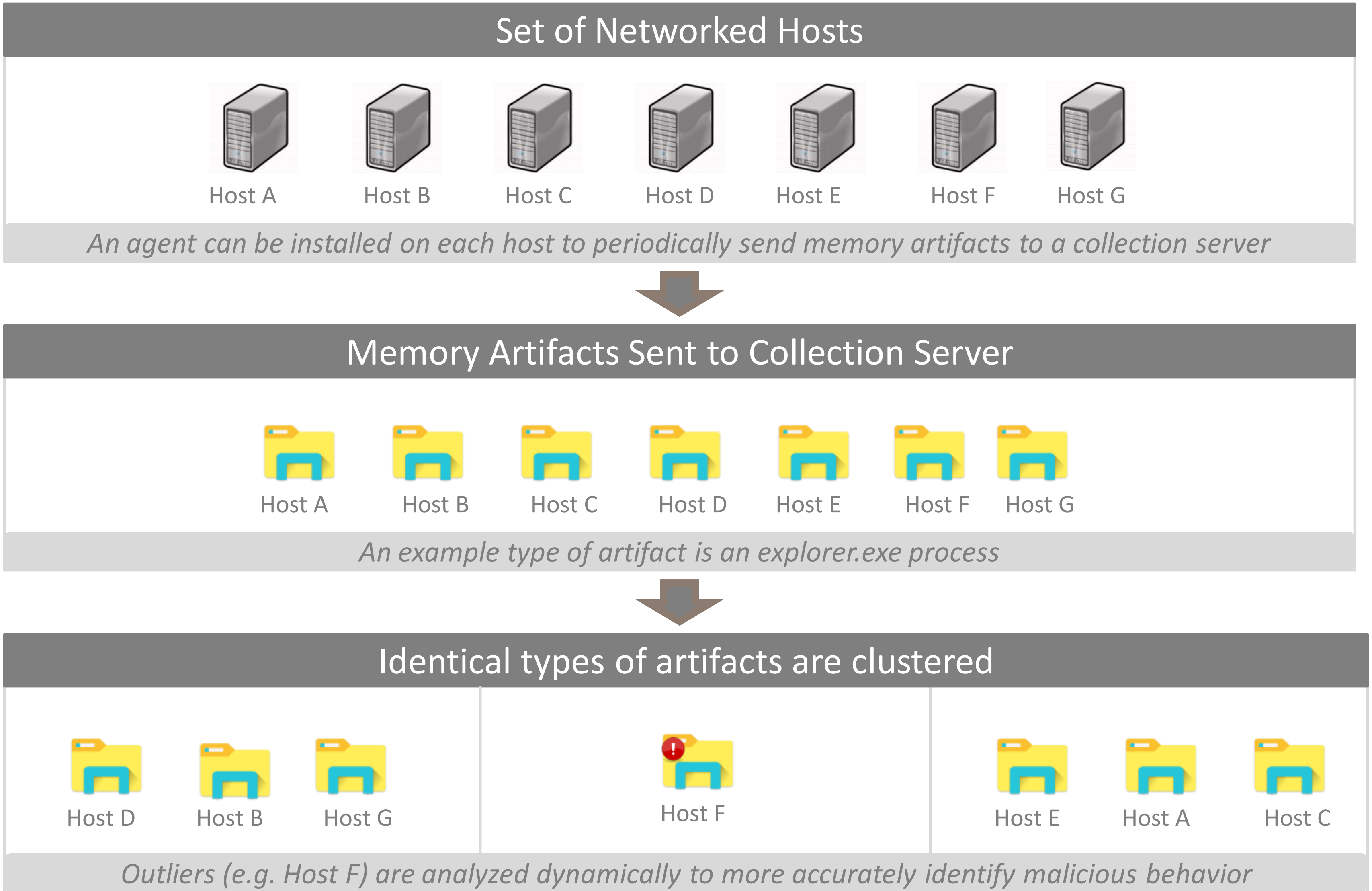
Main Takeaways



Main Takeaways

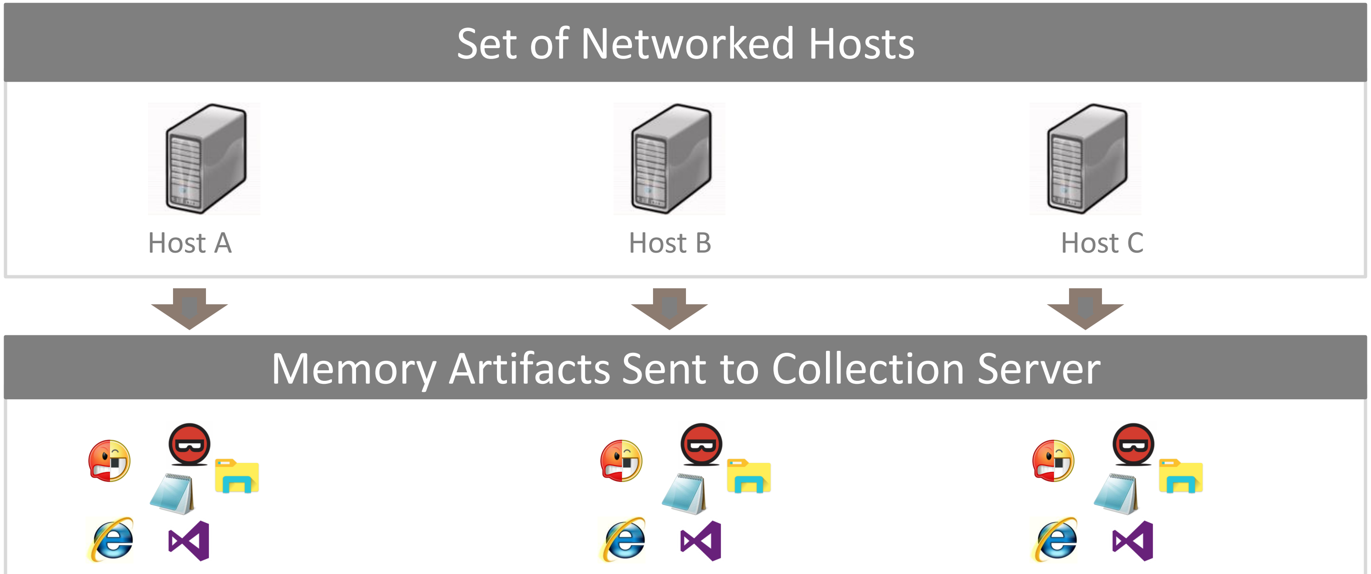


MAIN TAKEAWAYS



MEMORY ACQUISITION AUTOMATION

Memory Acquisition Automation



An example of process artifacts acquired from physical memory of each host

- **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.ex      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

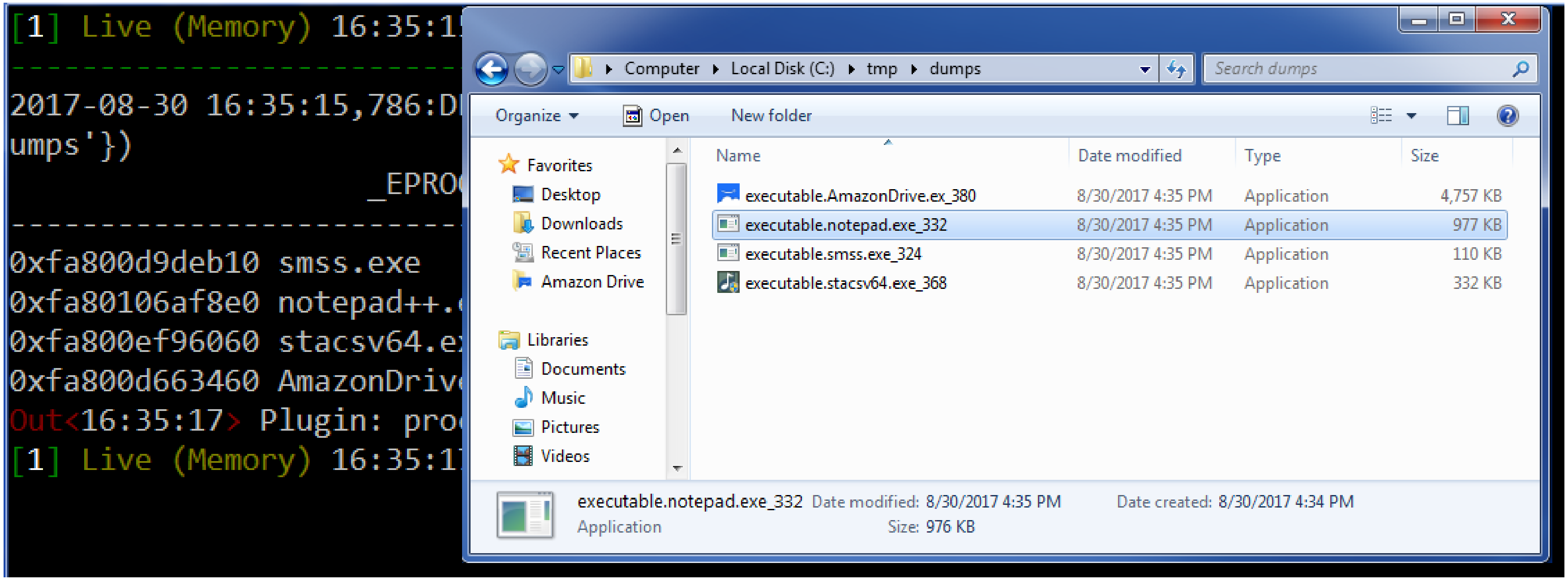
- Capturing live dumps of binaries w/ Rekall

```
[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
umps'})
                    _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*

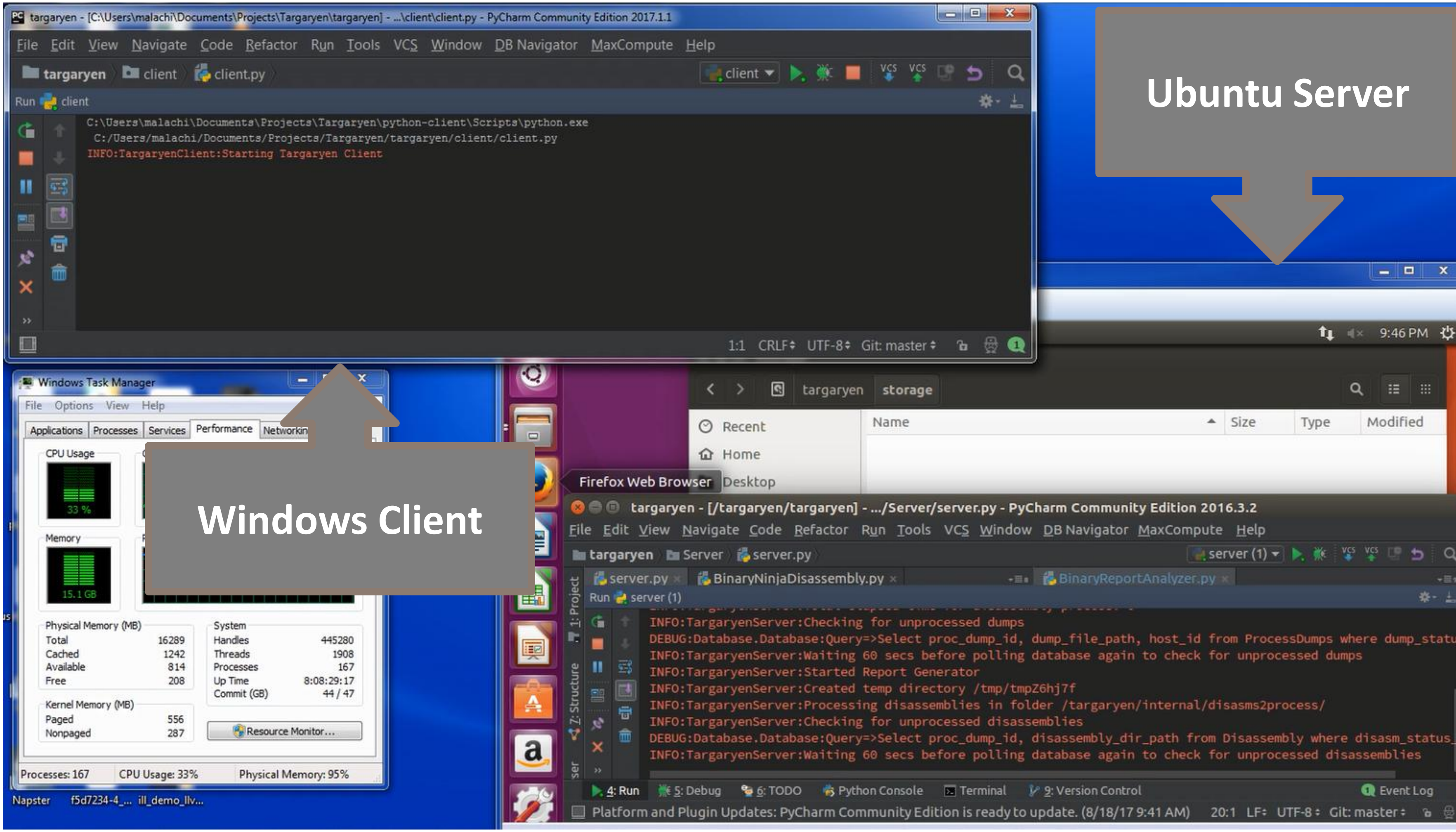
Memory Acquisition Automation

- Capturing live dumps of binaries w/ Rekall

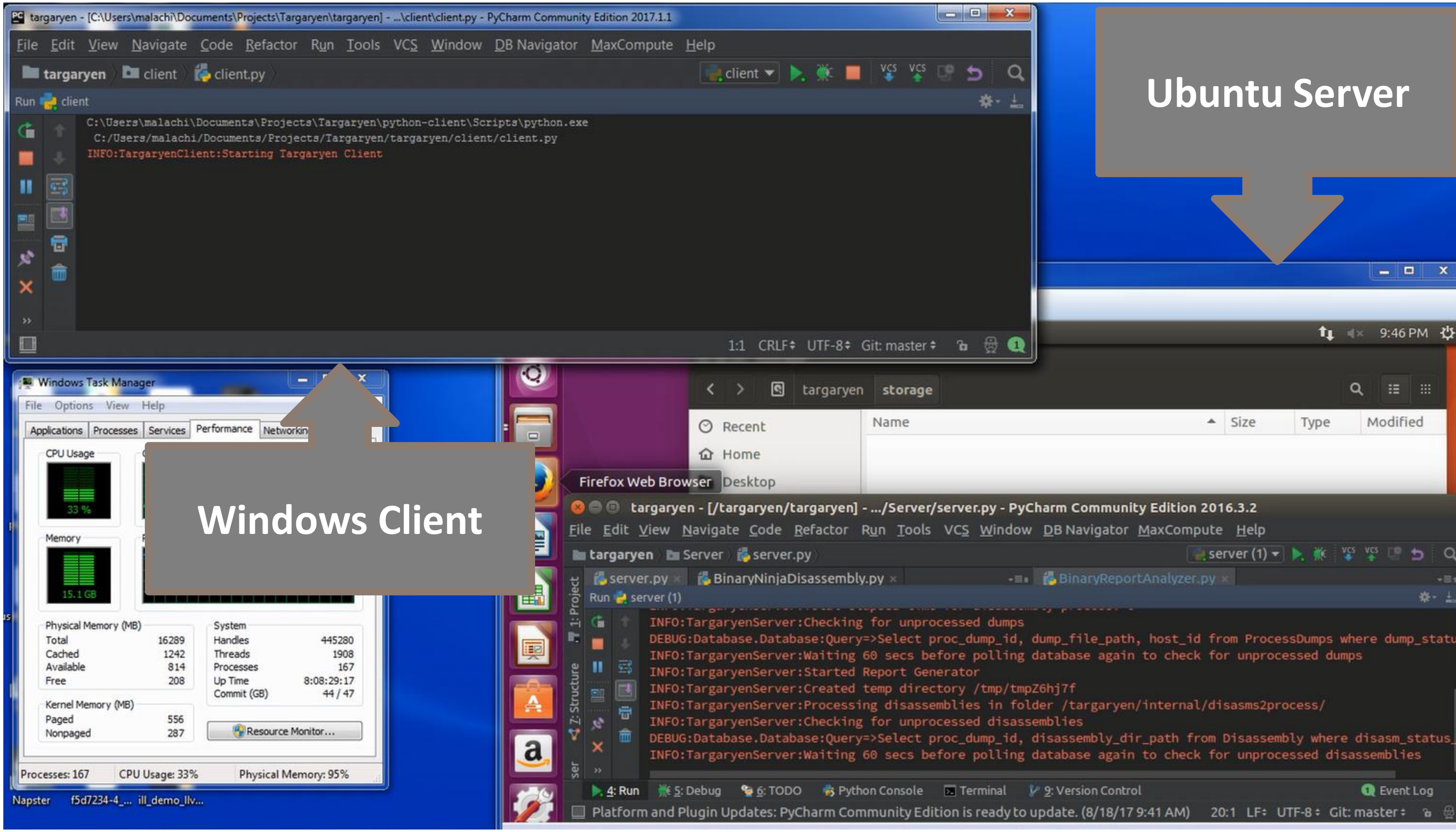


procdump *dumps a set of specified processes (given pids as input) to a desired directory*

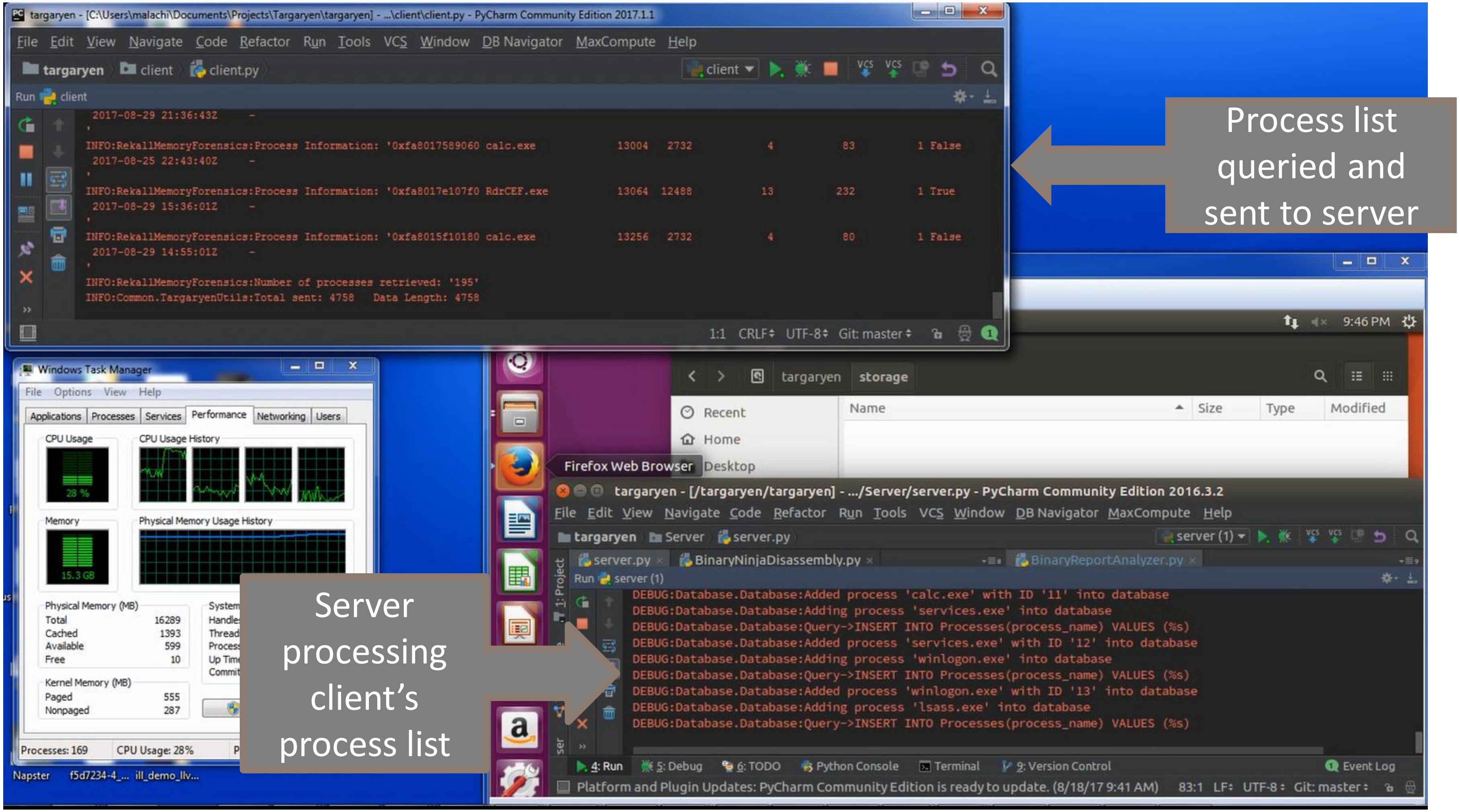
(Demo) Memory Acquisition Automation



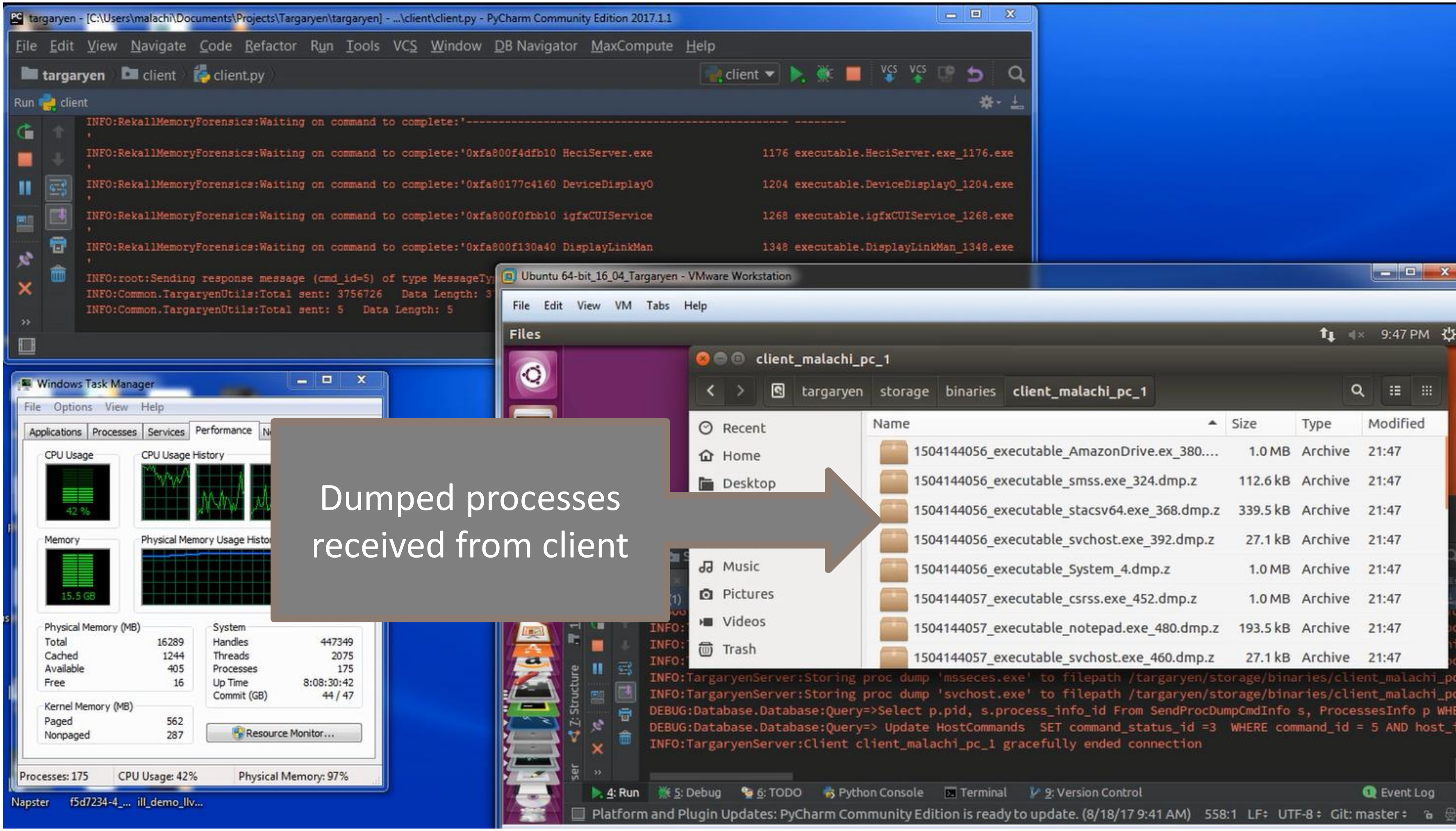
(Demo) Memory Acquisition Automation



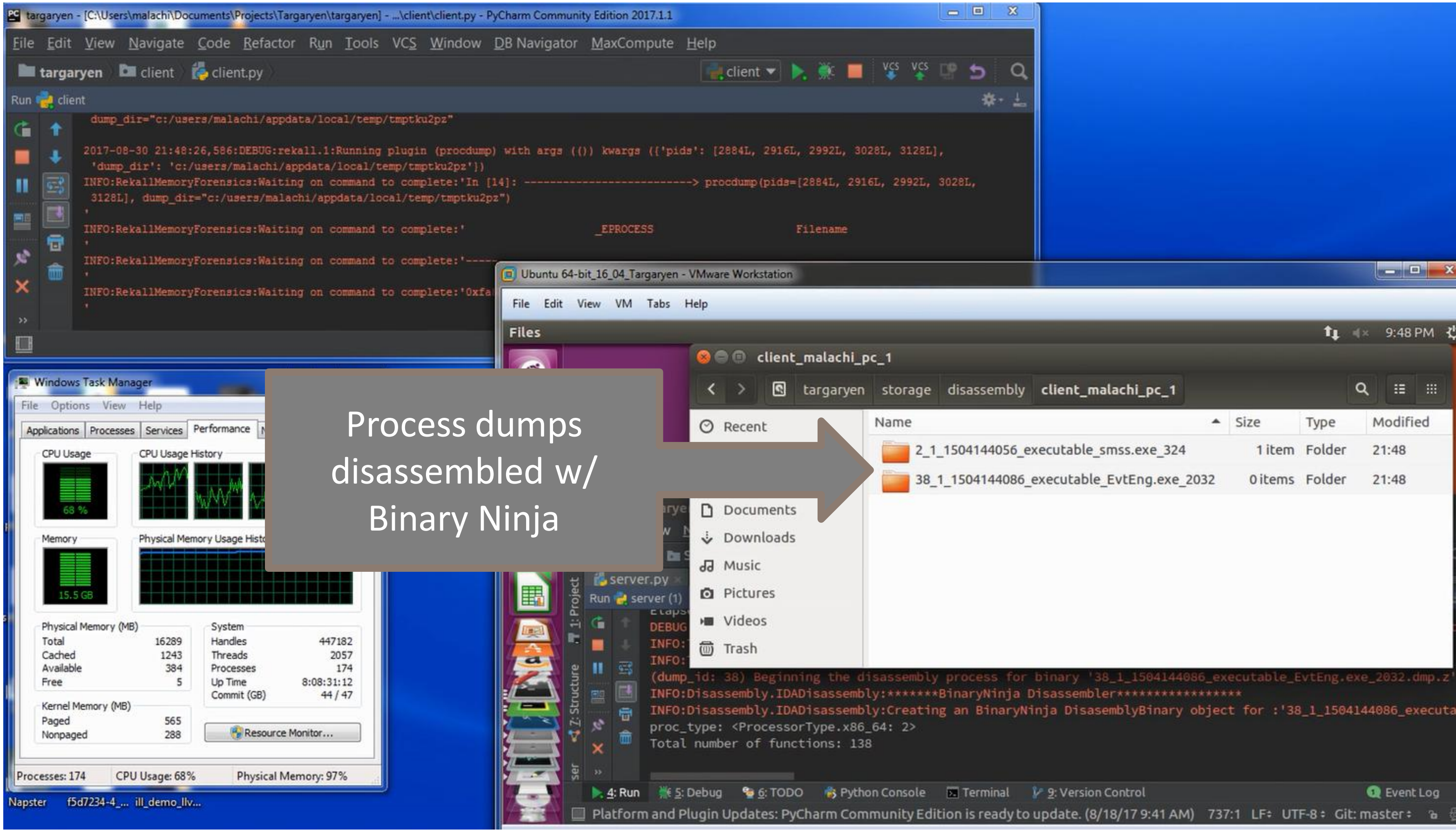
(Demo) Memory Acquisition Automation



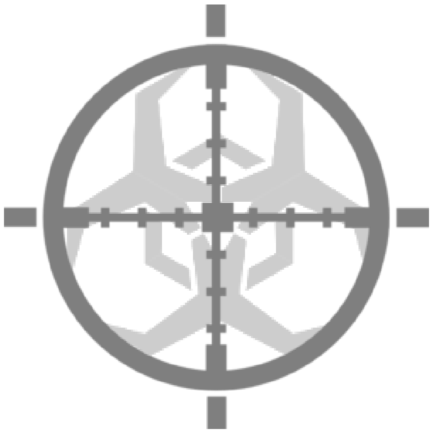
(Demo) Memory Acquisition Automation



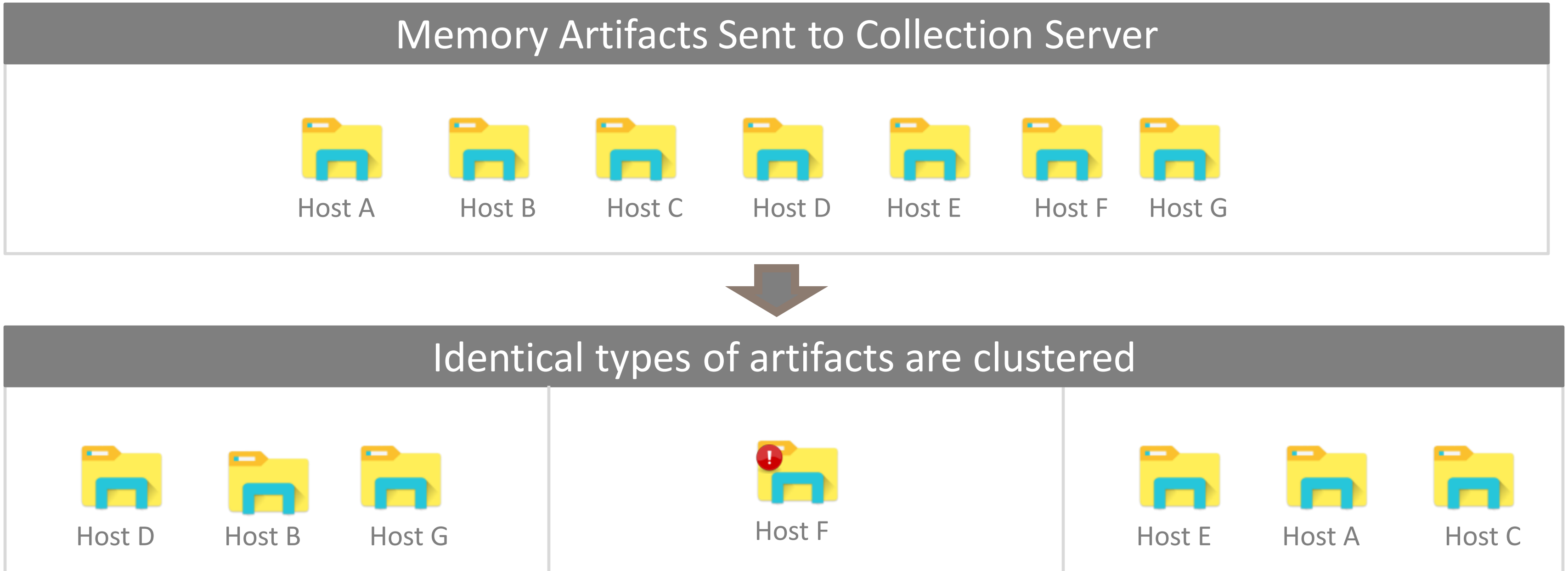
(Demo) Memory Acquisition Automation



PHASE I DETECTION: STATIC ANALYSIS

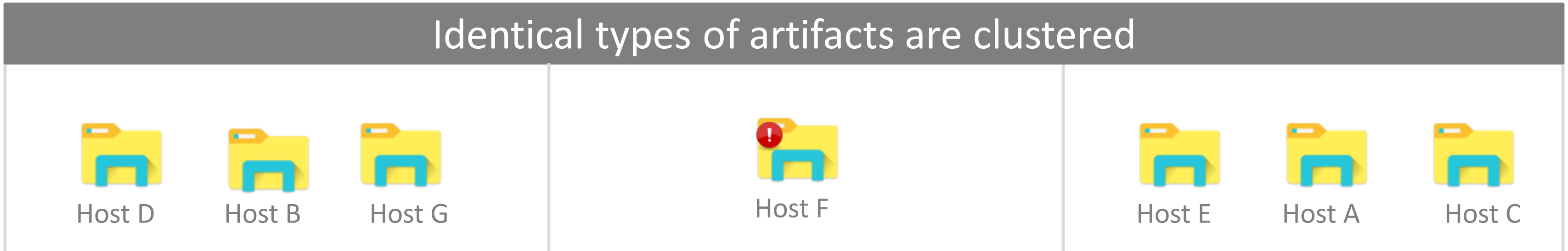


Phase I Detection: Static Analysis



- **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

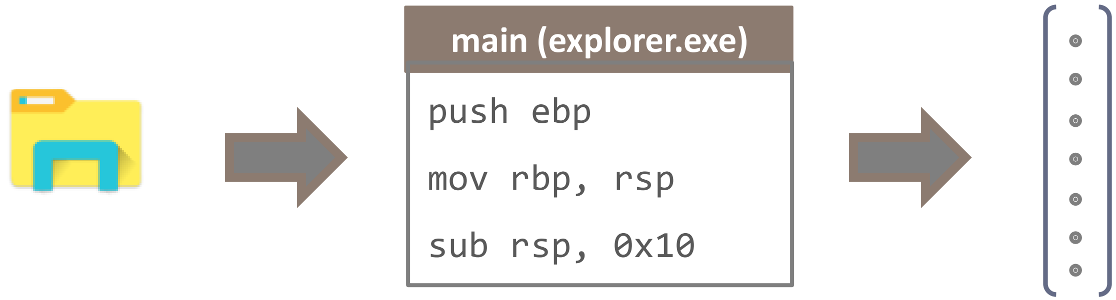
PHASE I DETECTION: STATIC ANALYSIS

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

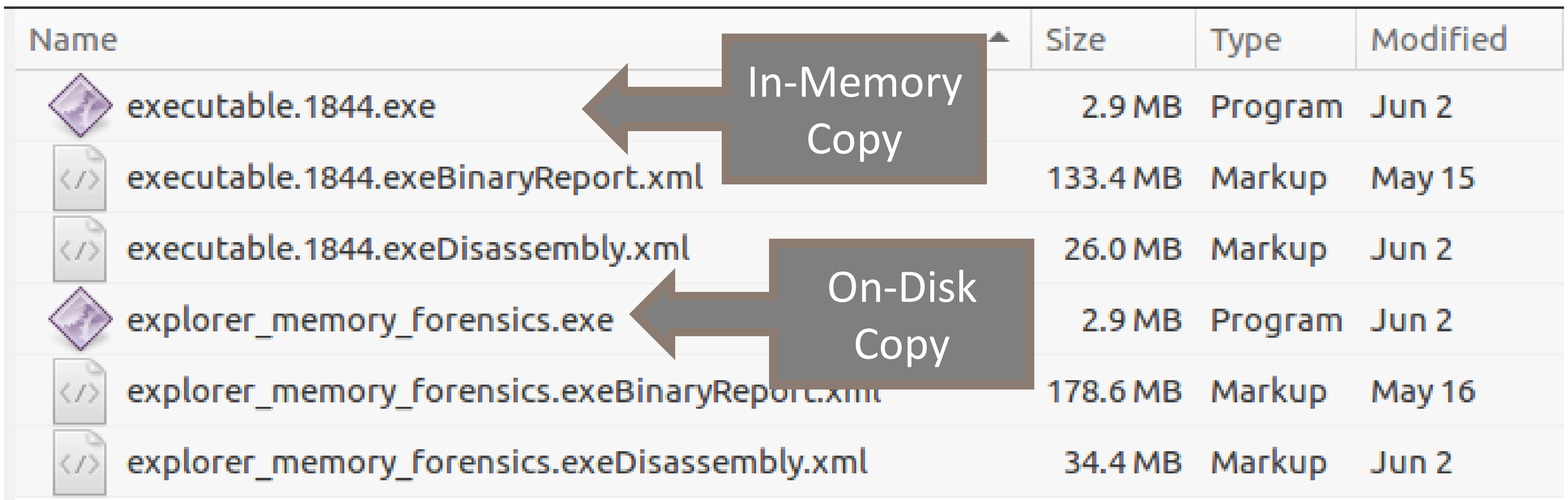
Step 2: Compute the similarity of a pair of vectors

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



Efficient Diffing Algorithm

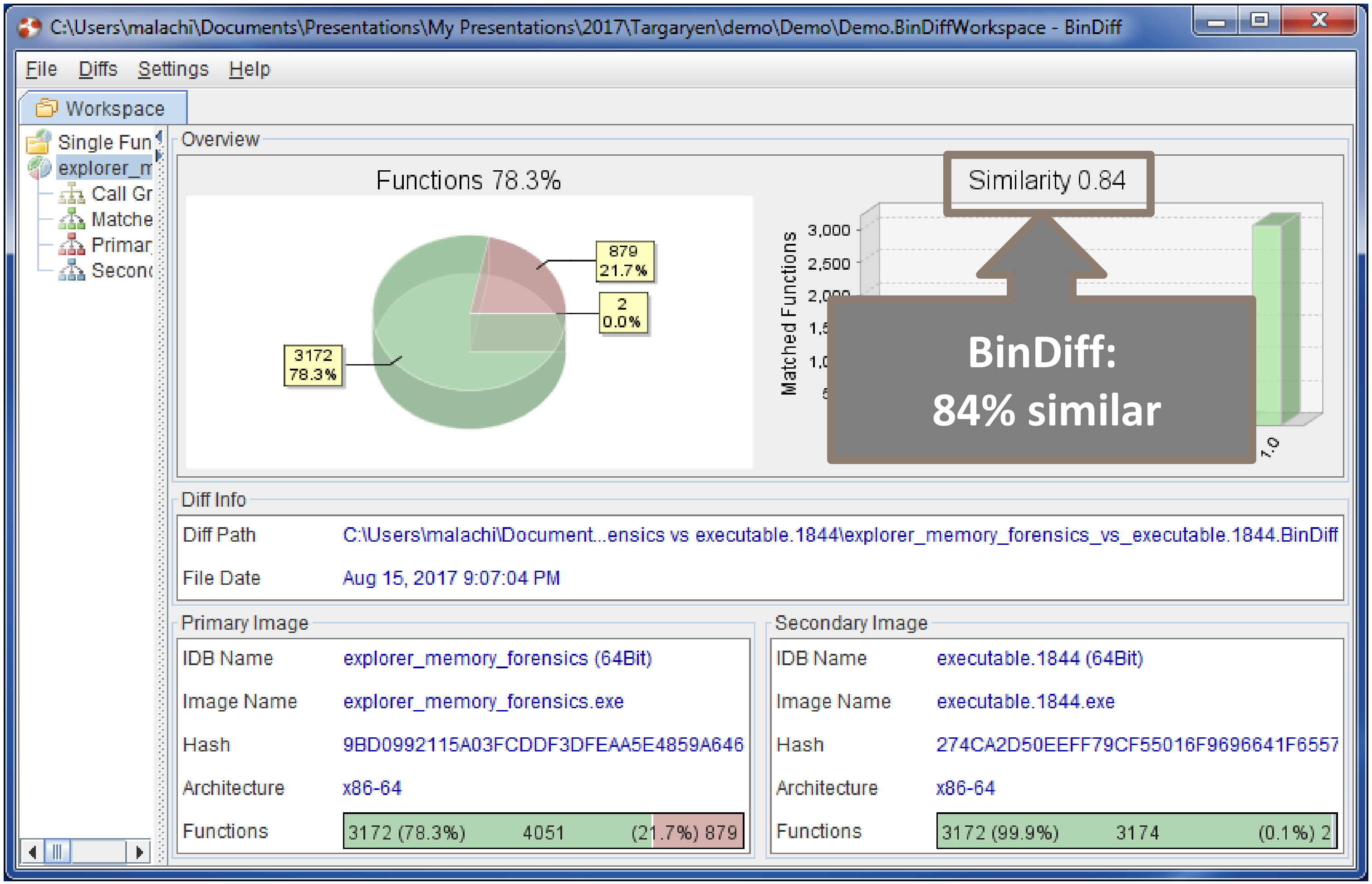
- Evaluating similarity function against explorer.exe

```
/home/targaryen/.virtualenvs/angr/bin/python /targaryen/targaryen/BinaryReportAnalyzer/ReportSimilarityAnalysis.
INFO:BinaryReportAnalyzer.ReportSimilarityAnalysis:
Report Similarity Analysis

INFO:BinaryReportAnalyzer.ReportAnalysis:
(Elasped Time: 0.0248849391937) Deserialized report analysis
    file 'executable.1844.exe'
    hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a6287044d17690a75bf628'

INFO:BinaryReportAnalyzer.ReportAnalysis:
(Elasped 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

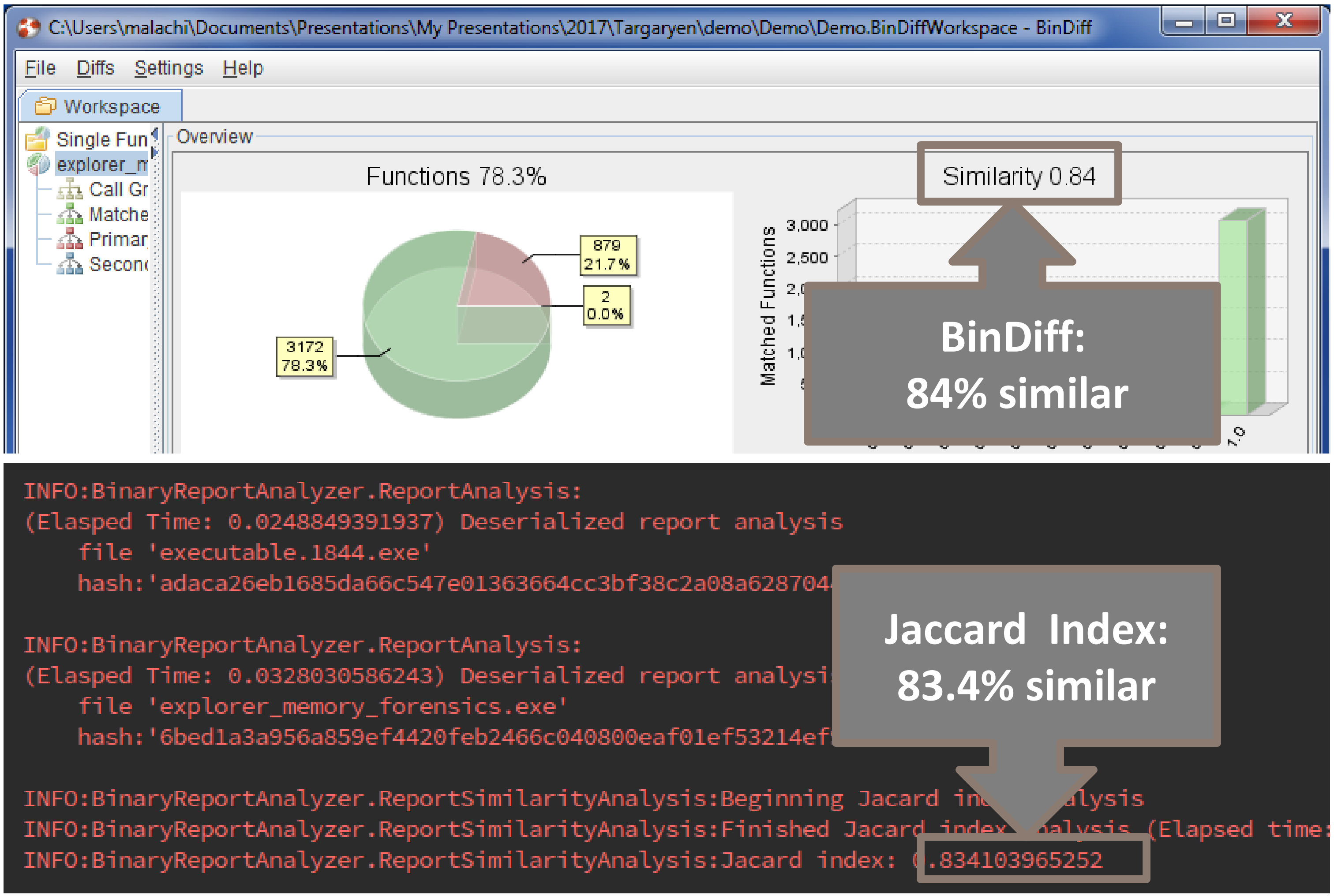
Efficient Diffing Algorithm

- Comparing similarity results against BinDiff



Efficient Diffing Algorithm

- Comparing similarity results against BinDiff



Efficient Diffing Algorithm

- Performance vs BinDiff

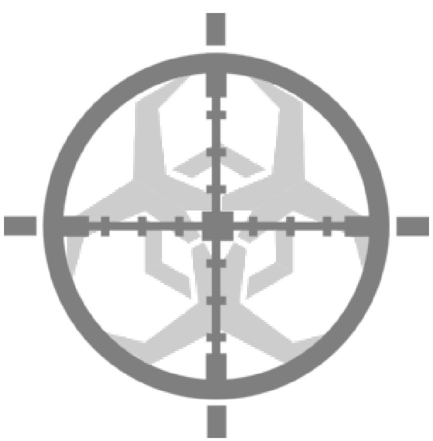
```
INFO:BinaryReportAnalyzer.ReportAnalysis:
(Elasped Time: 0.0248849391937) Deserialized report analysis
    file 'executable.1844.exe'
    hash:'adaca26eb1685da66c547e01363664cc3bf38c2a08a6287044d17690a75bf628'

INFO:BinaryReportAnalyzer.ReportAnalysis:
(Elasped 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

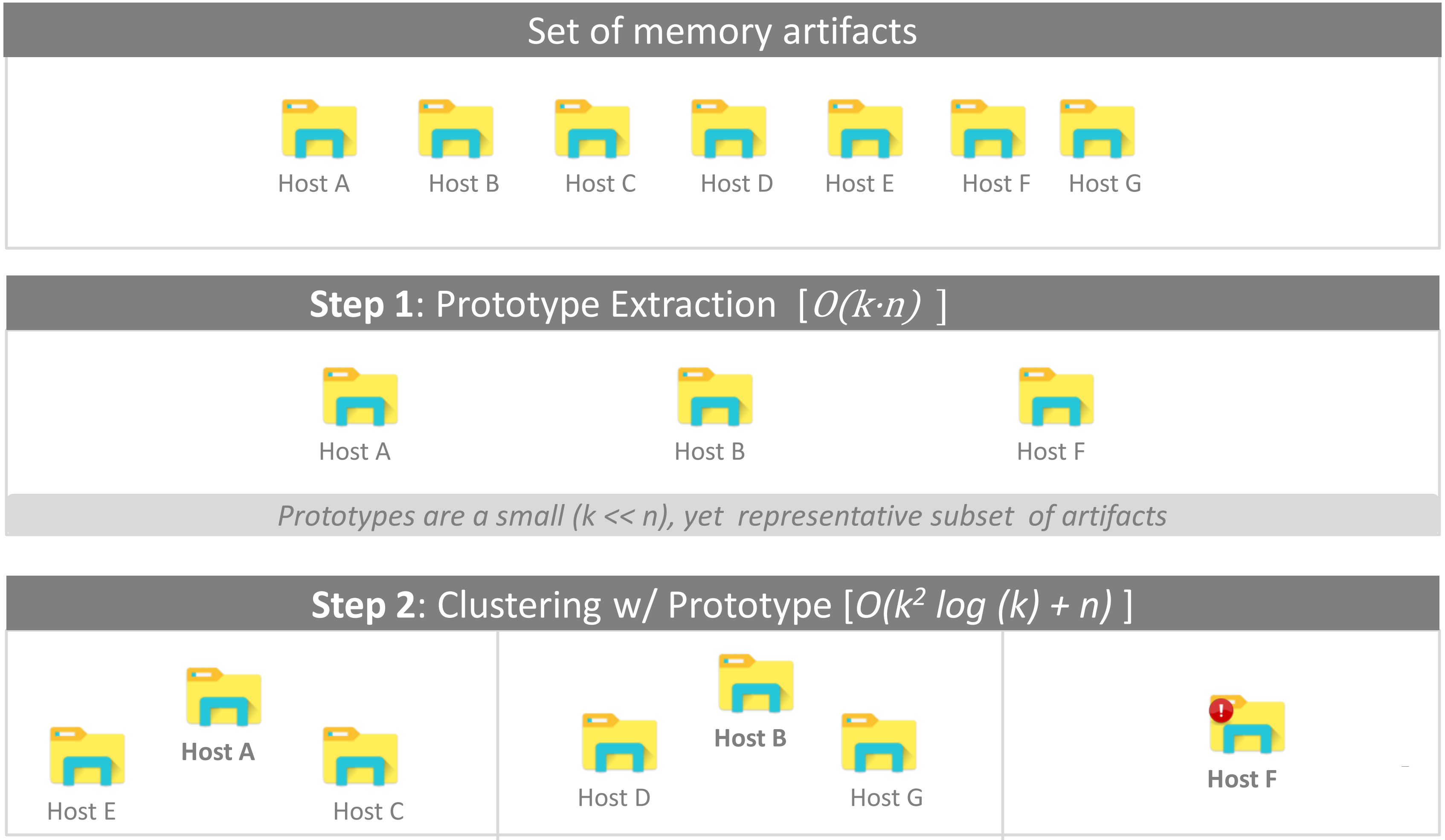


CLUSTERING ARTIFACTS AT SCALE

- 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 << n*

CLUSTERING ARTIFACTS AT SCALE

- **Approximate Clustering Algorithm [1] Sketch**



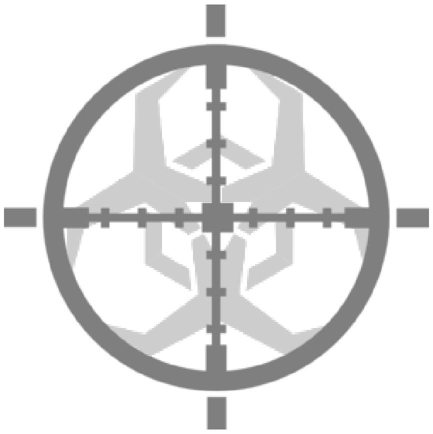
Clustering Artifacts At Scale

- Prototype Extraction Algorithm [1]

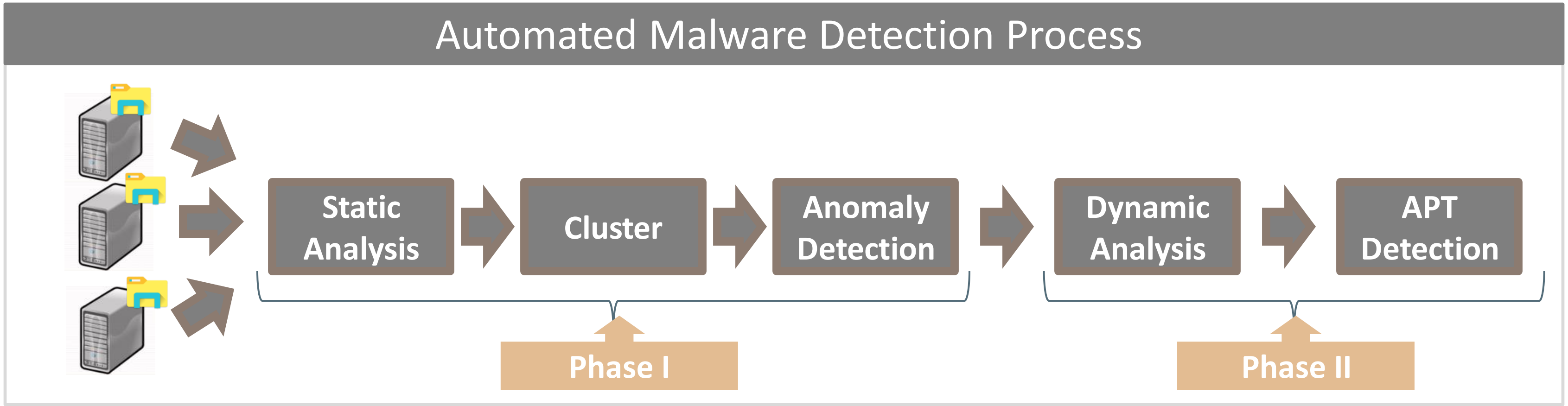
Algorithm 1 Prototype extraction

$prototypes \leftarrow \emptyset$
$distance[x] \leftarrow \infty$ for all $x \in reports$
while $\max(distance) > d_p$ **do**
    choose z such that $distance[z] = \max(distance)$
    for $x \in reports$ and $x \neq z$ **do**
        if $distance[x] > ||\hat{\varphi}(x) - \hat{\varphi}(z)||$ **then**
            $distance[x] \leftarrow ||\hat{\varphi}(x) - \hat{\varphi}(z)||$
    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***

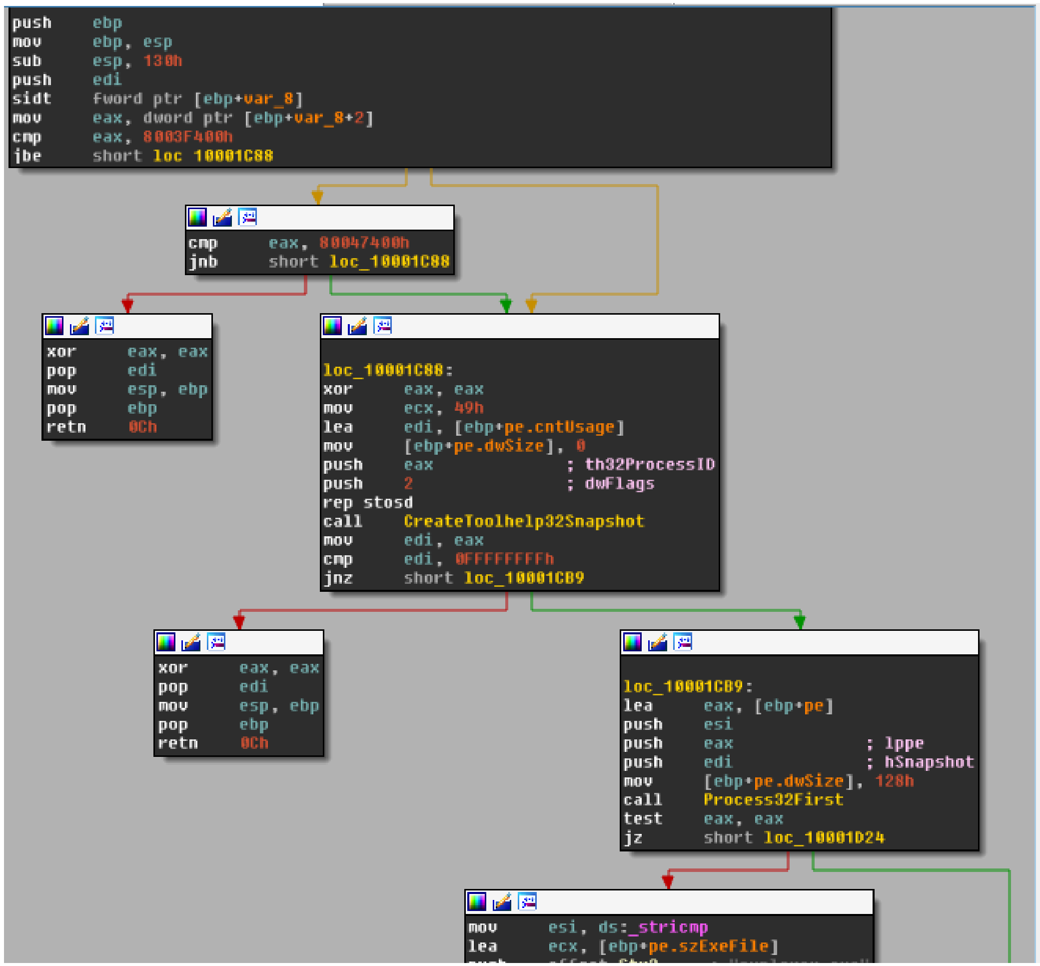
Phase II: Dynamic Analysis

- **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




```
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.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;
```

(**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.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;
```

Sample J Call Trace Example
memset(,0, 0x124u)
CreateToolhelp32Snapshot()
Process32First()
stricmp(., "explorer.exe")
Process32Next(.,.)
stricmp(., "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)`

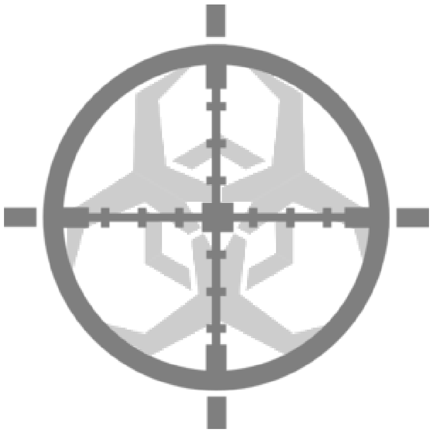
- **Call Trace Analysis**
  1. Iterate through the process handles to see if a process with name "explorer.exe" exists *(Check if user logged in)*
  2. If process exists, create a thread that infects target system

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

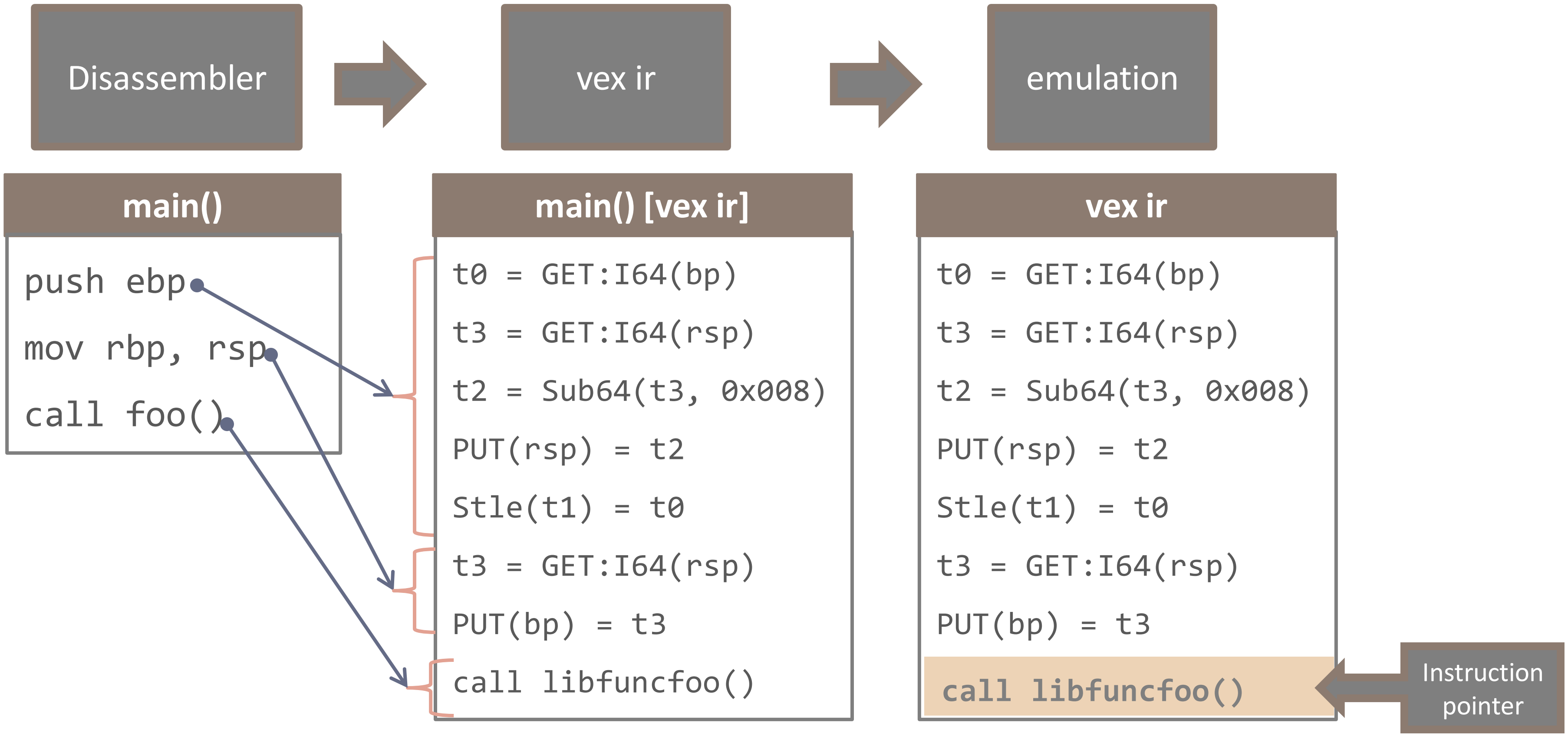
PHASE II DETECTION



AUTOMATING CALL TRACE GENERATION

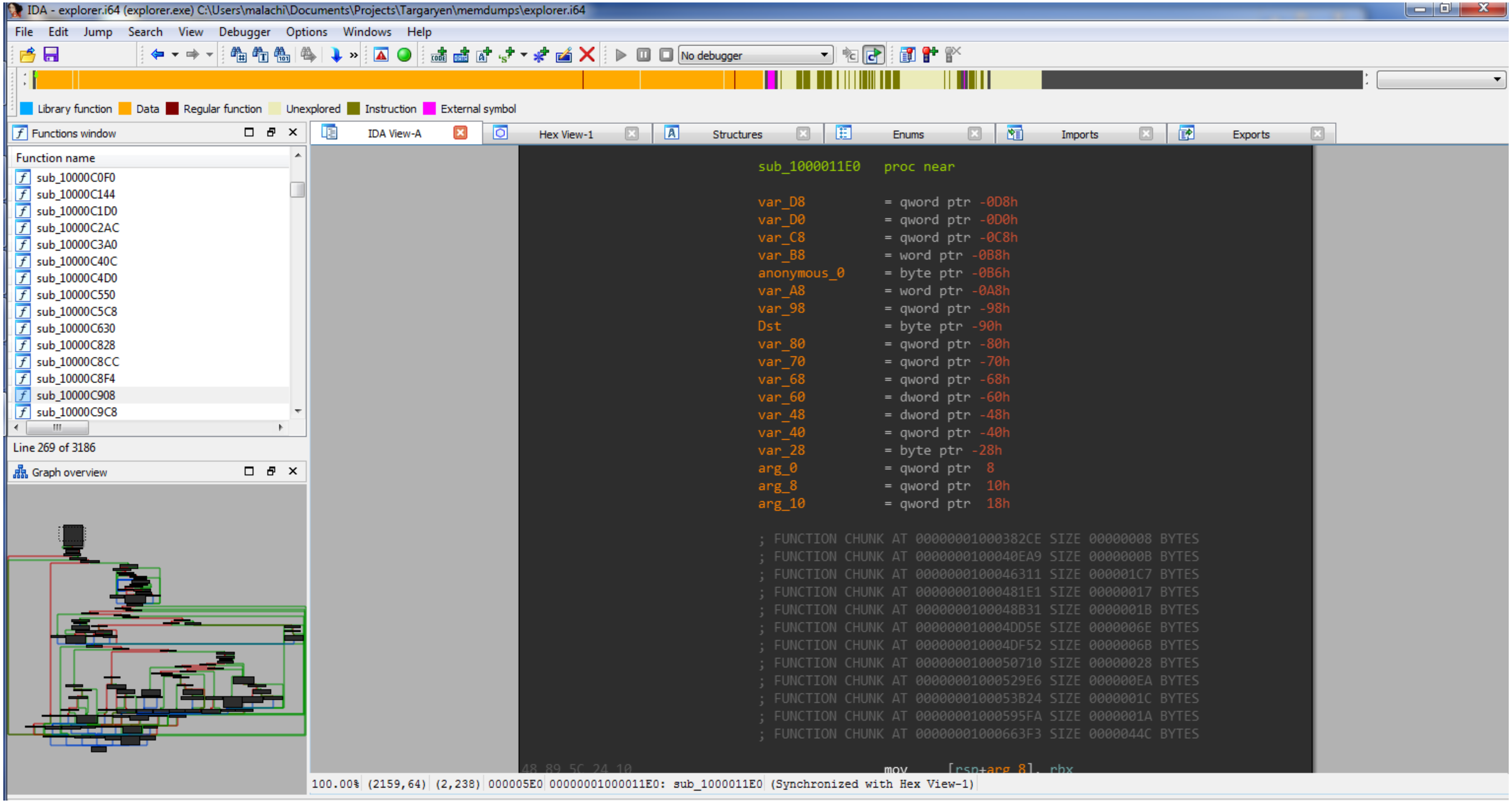
- Call Trace Generation
  1. **Load** the dumped binary into a disassembler (e.g. IDA or Binary Ninja) and extract the executable portion of binary
  2. **Lift** the executable portion of binary to vex, a RISC-like intermediate representation (ir) language
  3. **Perform** emulation on the vex ir to traverse unique code paths that originate from a specified entry function
  4. **Record** calls that occur during traversal of a code path

Automating Call Trace Generation

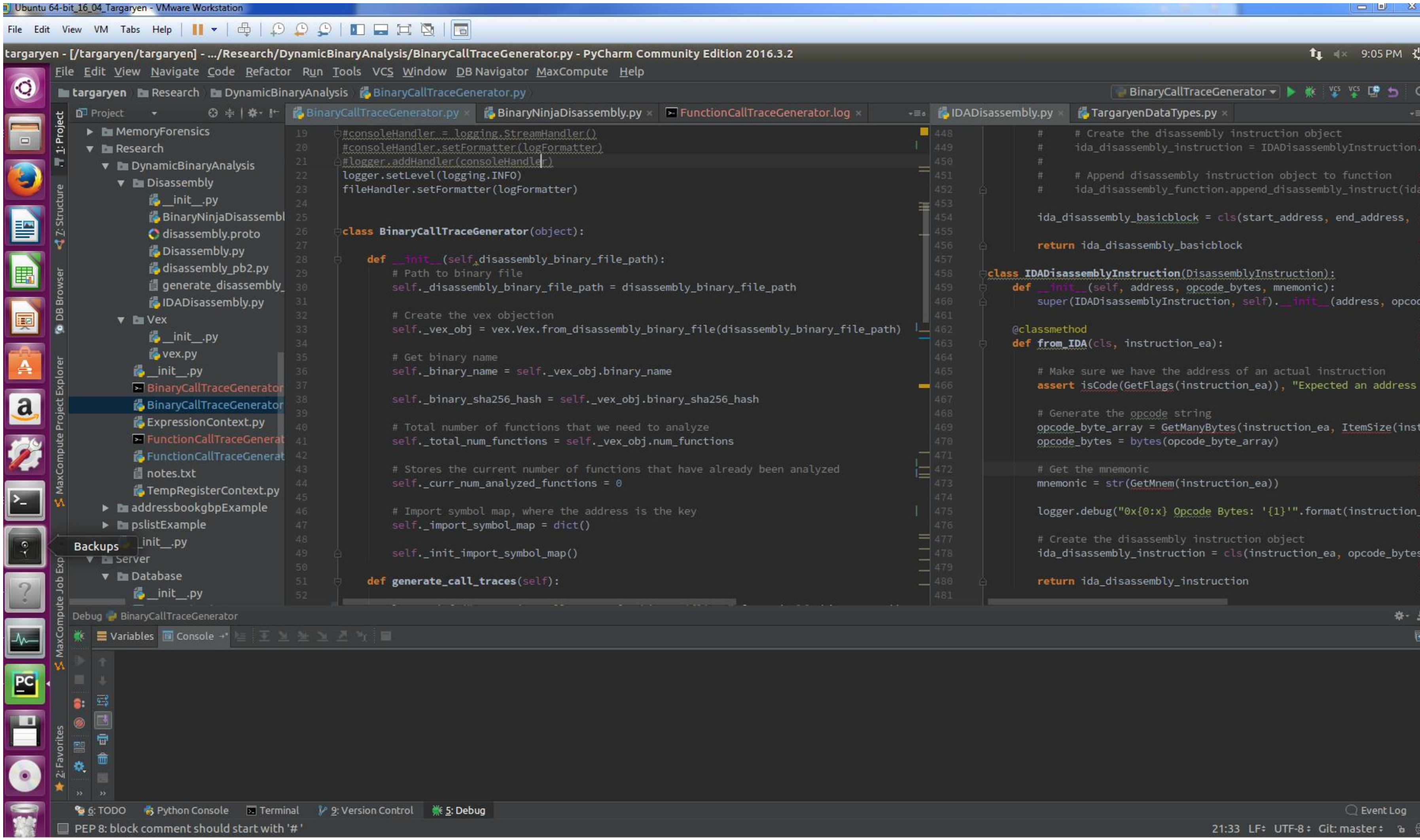


(Demo) Call Trace Generation

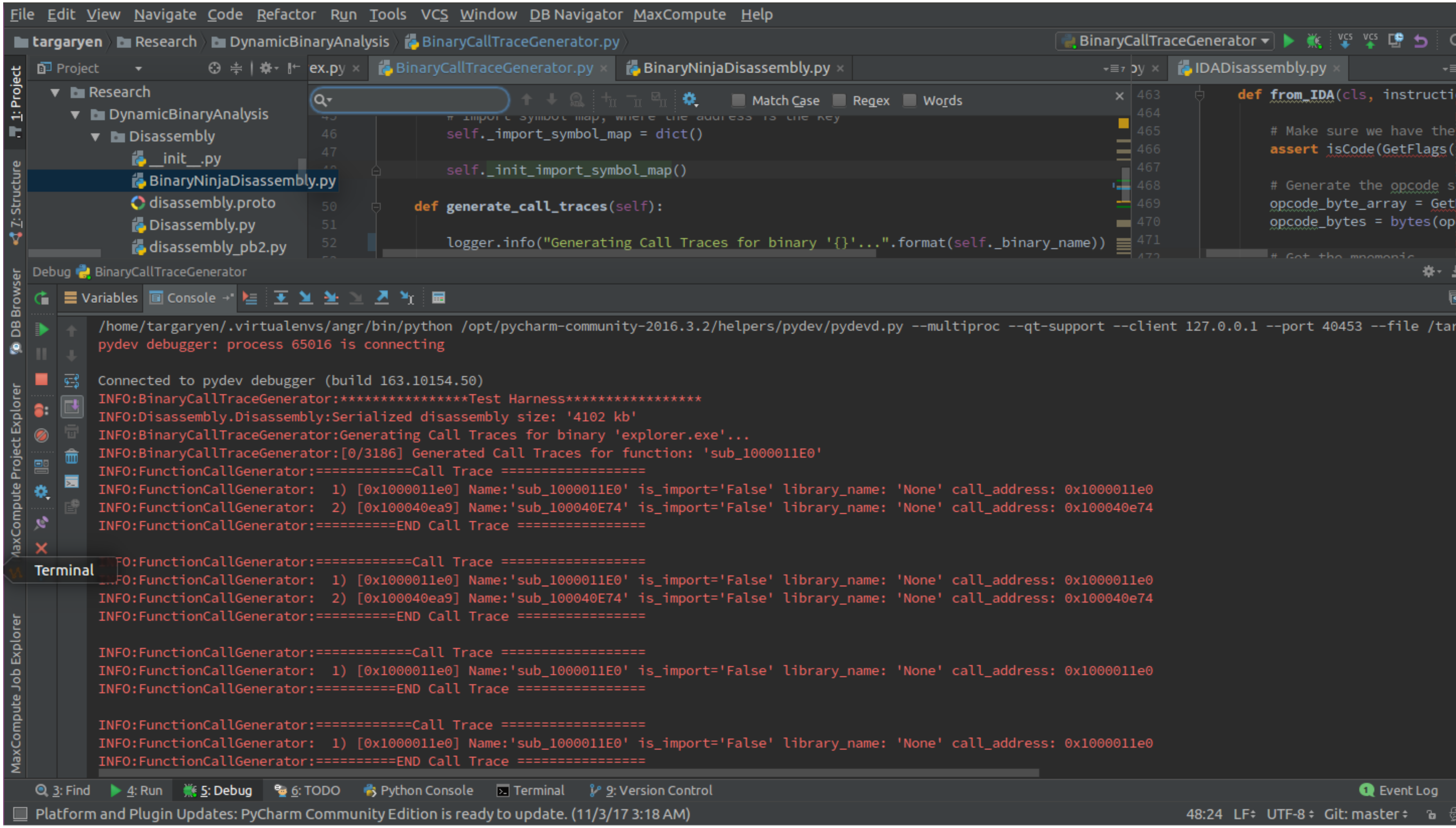
Target Binary: explorer.exe



(Demo) Call Trace Generation



(Demo) Call Trace Generation



(Demo) Call Trace Generation

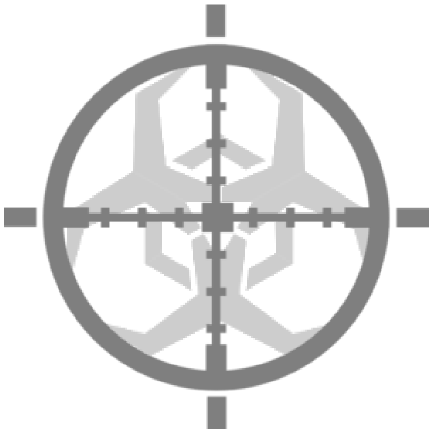
```
INFO:FunctionCallGenerator:  19) [0x100093652] Name:'GetWindowLongW' is_import='True' library_name: 'USER32' call_address: 0x1000bae
INFO:FunctionCallGenerator:  20) [0x100093672] Name:'SelectObject' is_import='True' library_name: 'GDI32' call_address: 0x1000ba748
INFO:FunctionCallGenerator:==========END 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:'Ge   arent' is_import='True' library_name: 'USER32'   l_address: 0x1000bab38
INFO:FunctionCallGenerator:  5) [0x10006670a] Name:'   n_QueryServiceExec' is_import='True' libra   : 'SHLWAPI' call_address
INFO:FunctionCallGenerator:  6) [0x10   ort='False' l   100047814
INFO:FunctionCallGenerator:  7) [0x10   'True' librar   0ba968
INFO:FunctionCallGenerator:  8) [0x10   ort='False' l   100050b80
INFO:FunctionCallGenerator:  9) [0x10   ' library_nam
INFO:FunctionCallGenerator:  10) [0x1000   is_import='False'   call_address: 0x10005087c
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
```

Import Library Function

Library name

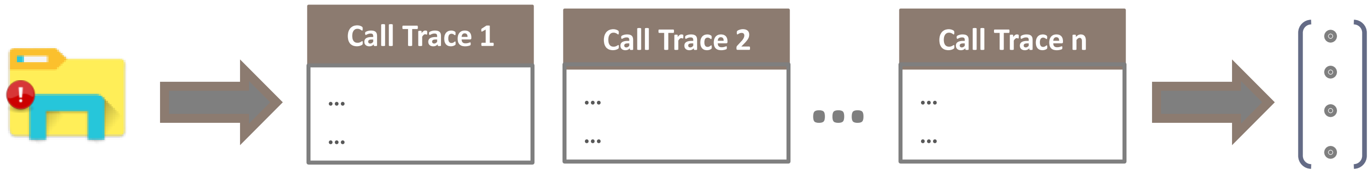
Leverage Call Trace Info to Detect APTs

PHASE II DETECTION



Leverage Call Trace Info to Detect APTs

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



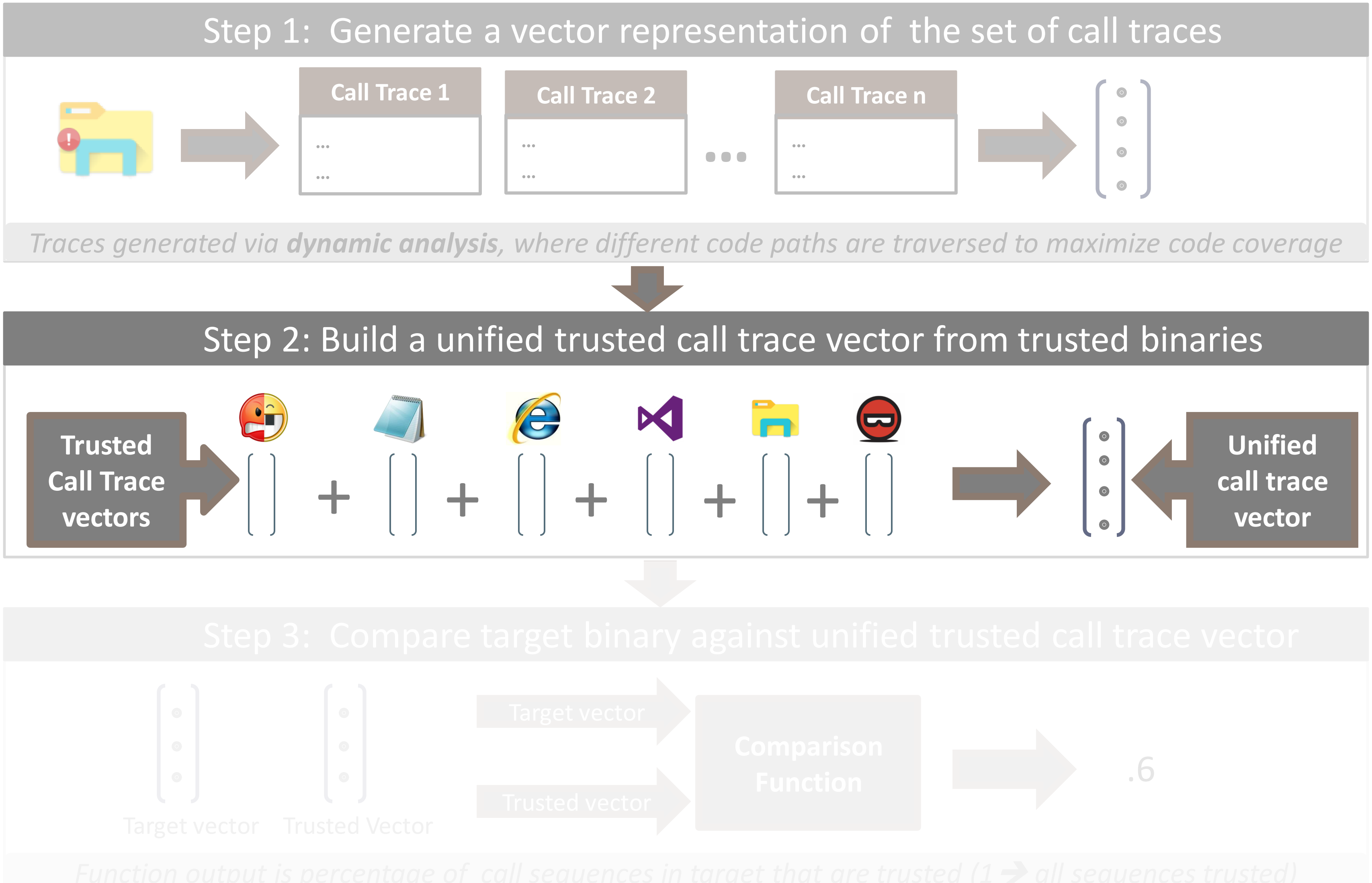
*Traces generated via **dynamic analysis**, where different code paths are traversed to maximize code coverage*

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

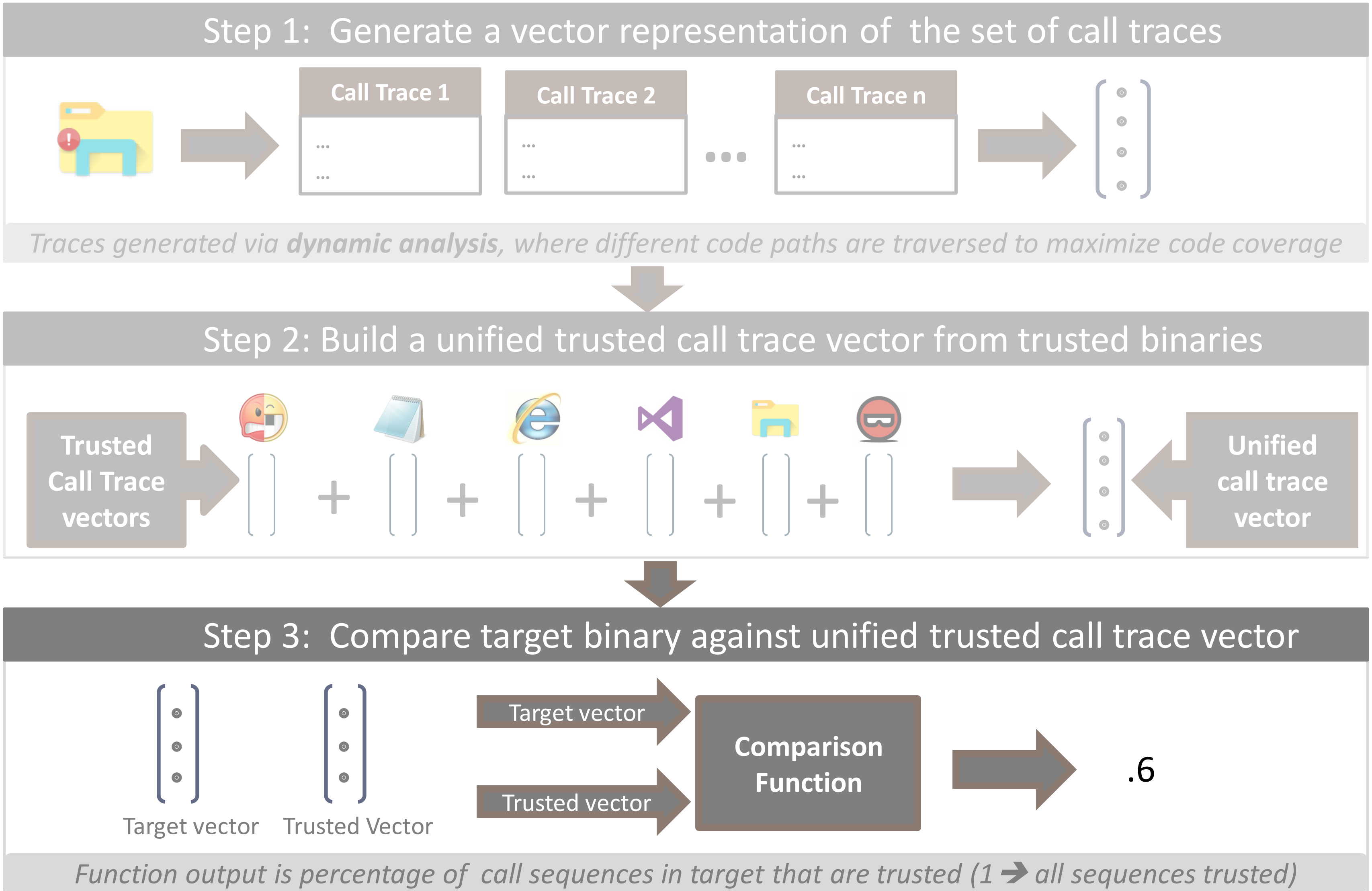
Step 3: Compare target binary against unified trusted call trace vector

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

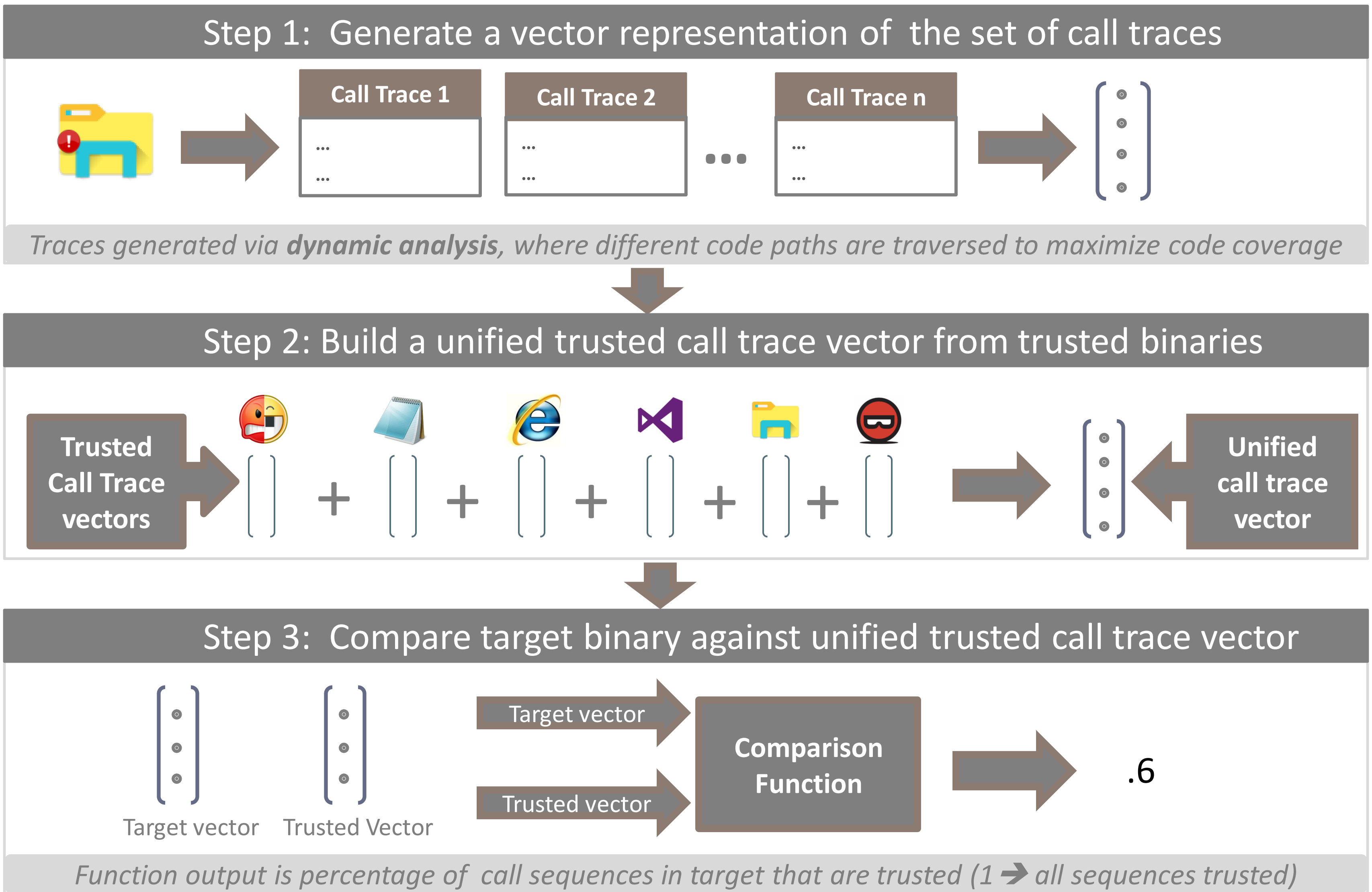
Leverage Call Trace Info to Detect APTs



Leverage Call Trace Info to Detect APTs



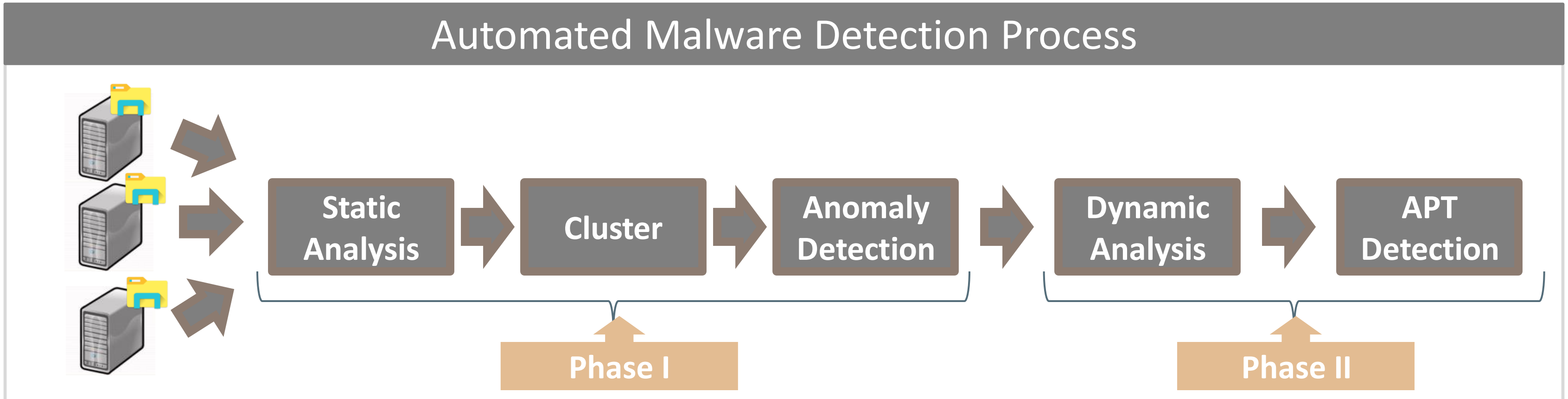
Leverage Call Trace Info to Detect APTs



(See Appendix E for more details about the algorithm)

CONCLUSION

Conclusion



Dynamic analysis is leveraged to reduce false positives (from Phase I) and to more accurately identify APTs

- **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
    1. The hosts ***are not*** under ***significant*** memory pressure
    2. 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

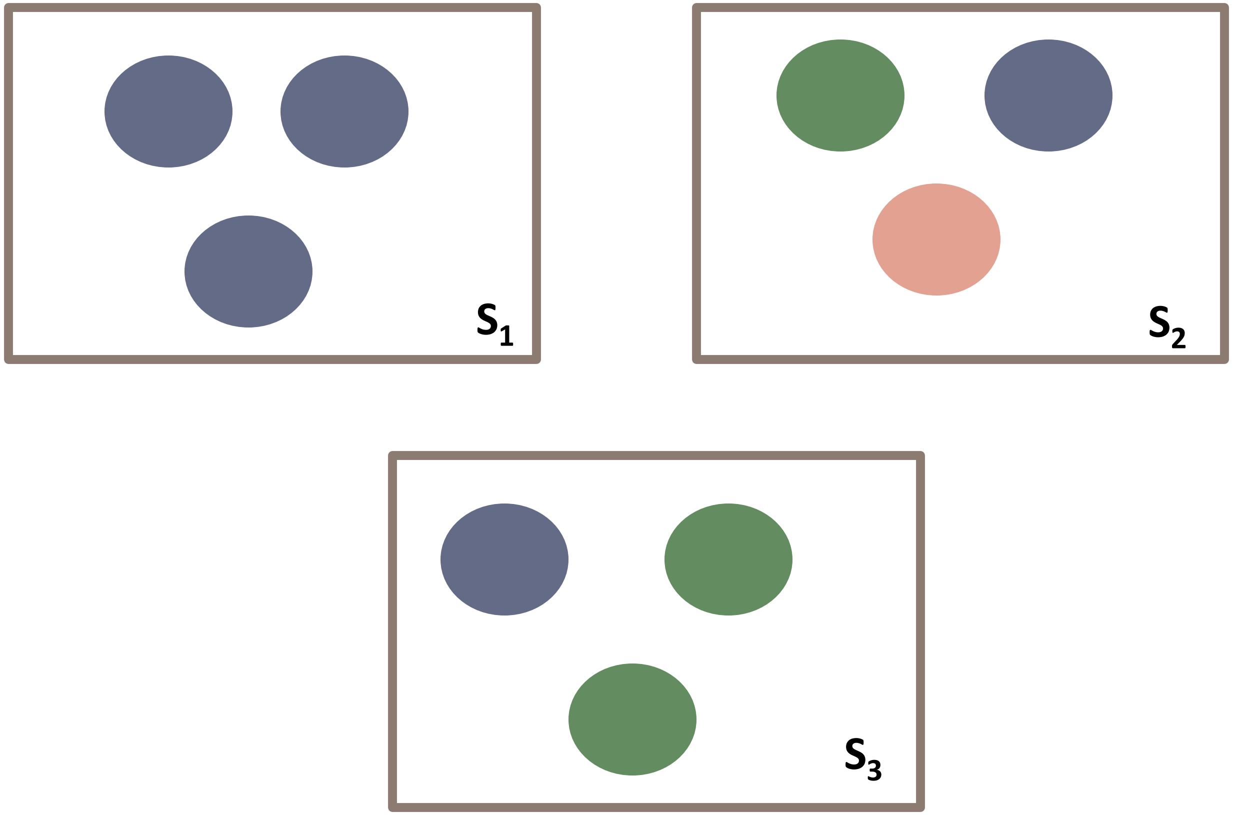
MACHINE LEARNING

- 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

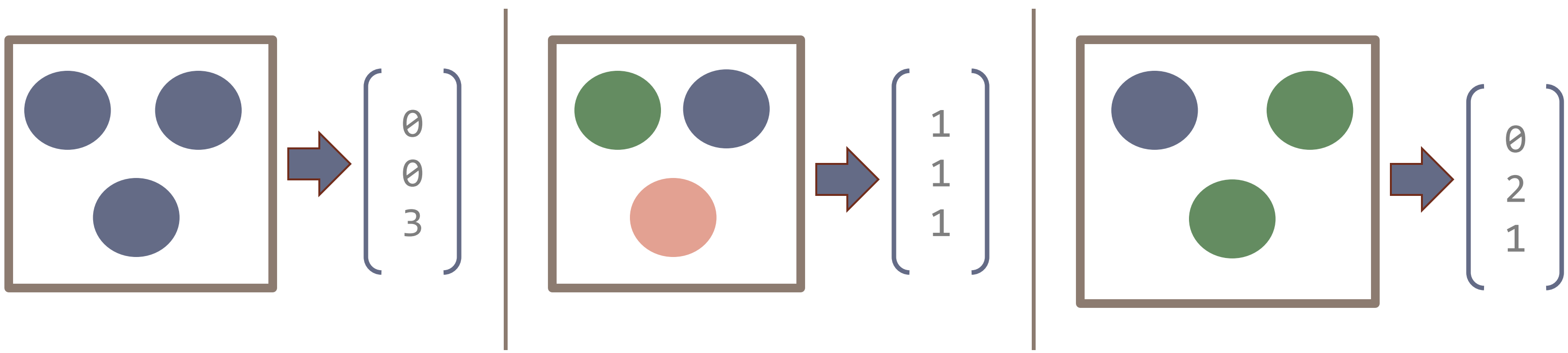


1. Representing Observations Mathematically

- 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



$$\begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix}$$

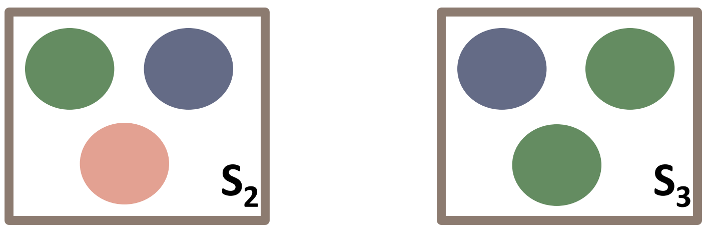
Machine Learning

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:

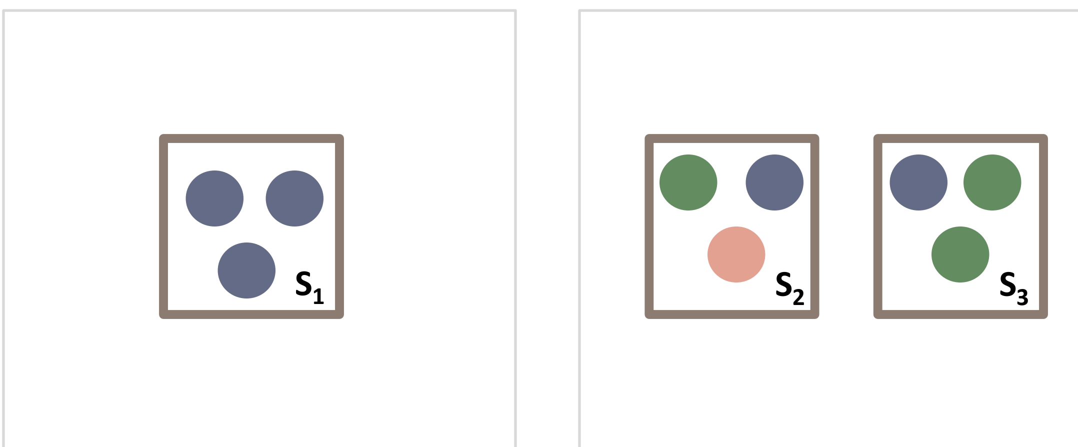


$$J(S_2, S_3) = \frac{S_2 \cap S_3}{S_2 \cup S_3} = \frac{2}{3}$$

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



APPENDIX B

Binary Analysis

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

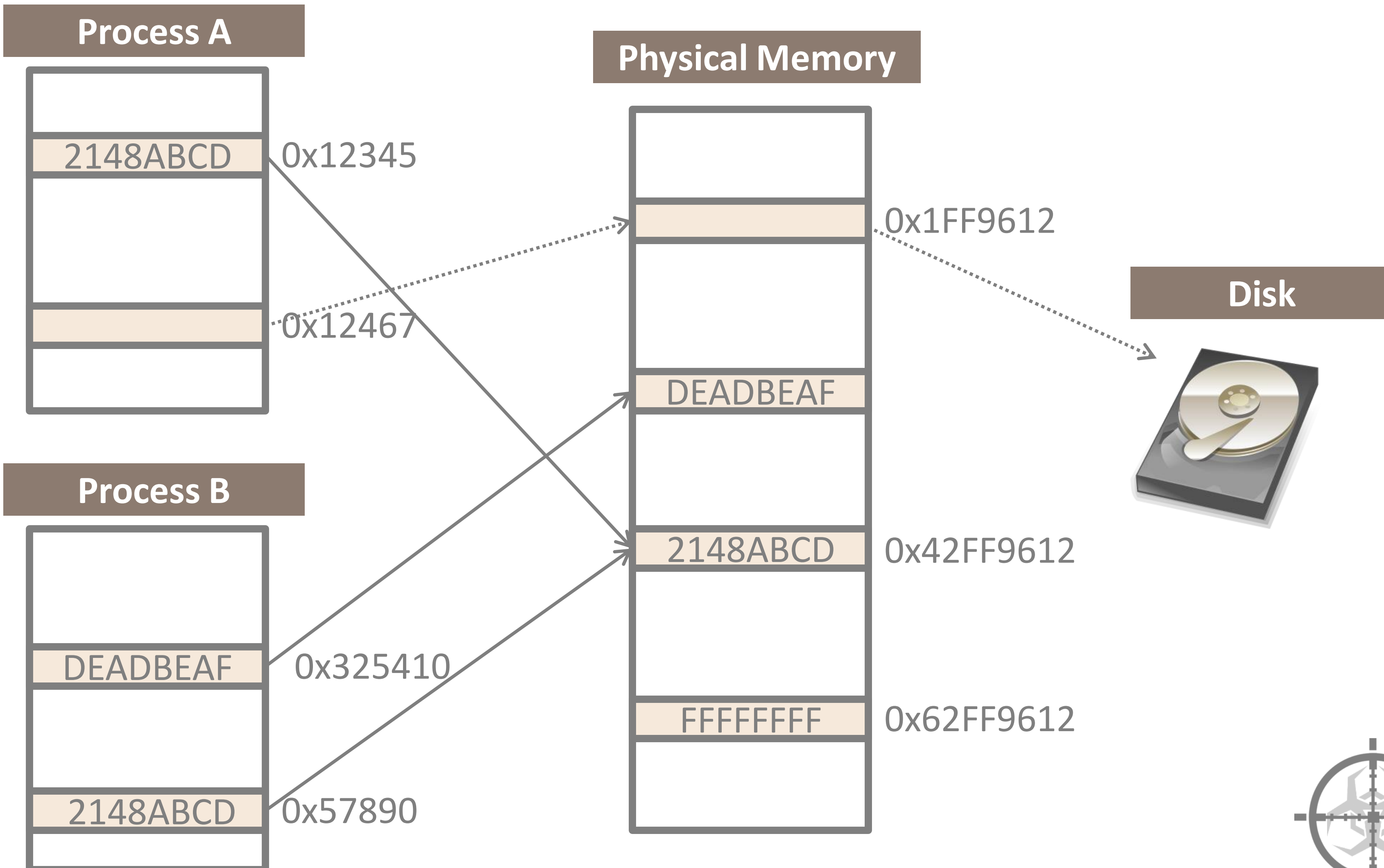
APPENDIX C

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

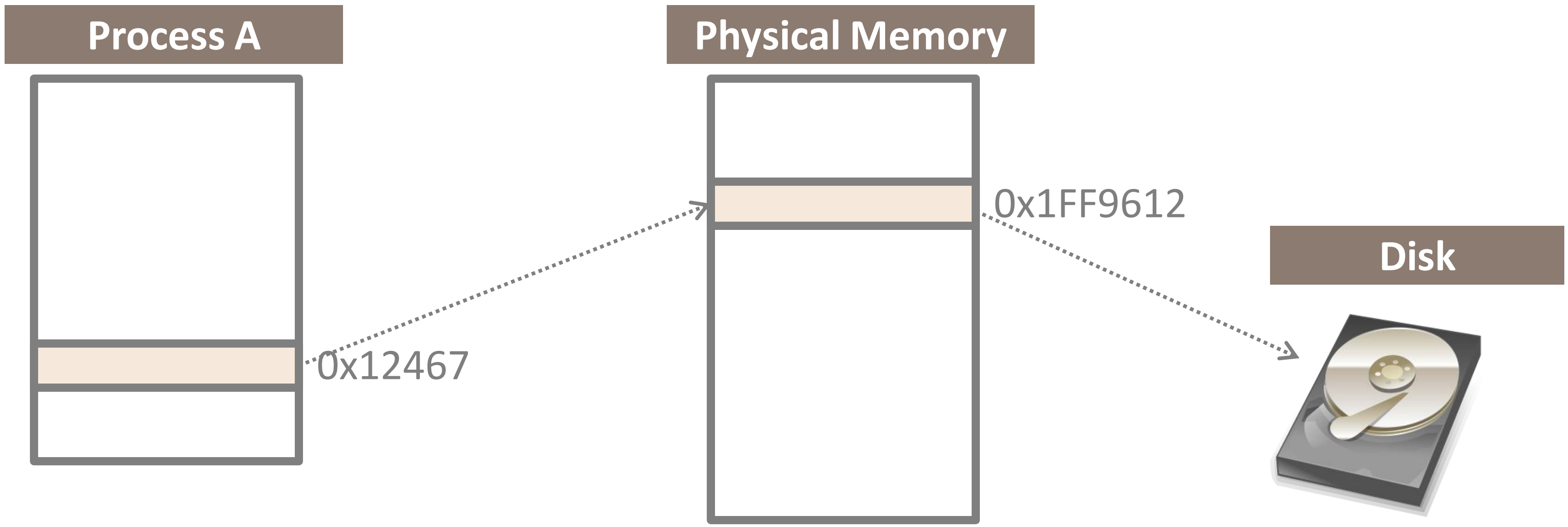
Memory Forensics

- 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***

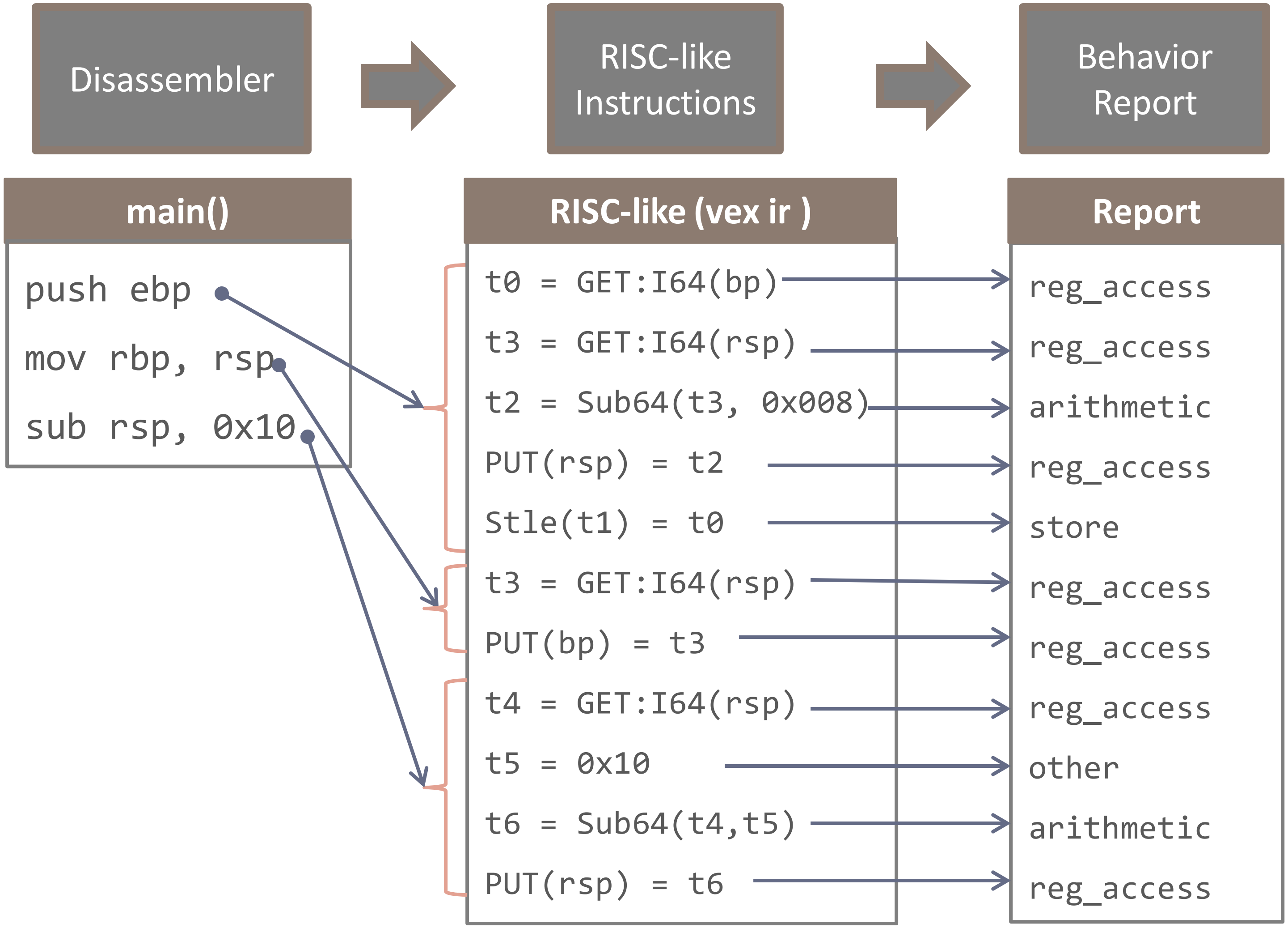
Generating a Vector Representation of a Binary

APPENDIX D

Binary Vector Generation

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

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

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

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	SEQ 2	SEQ 3
reg_access	reg_access	arithmetic
reg_access	arithmetic	reg_access
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

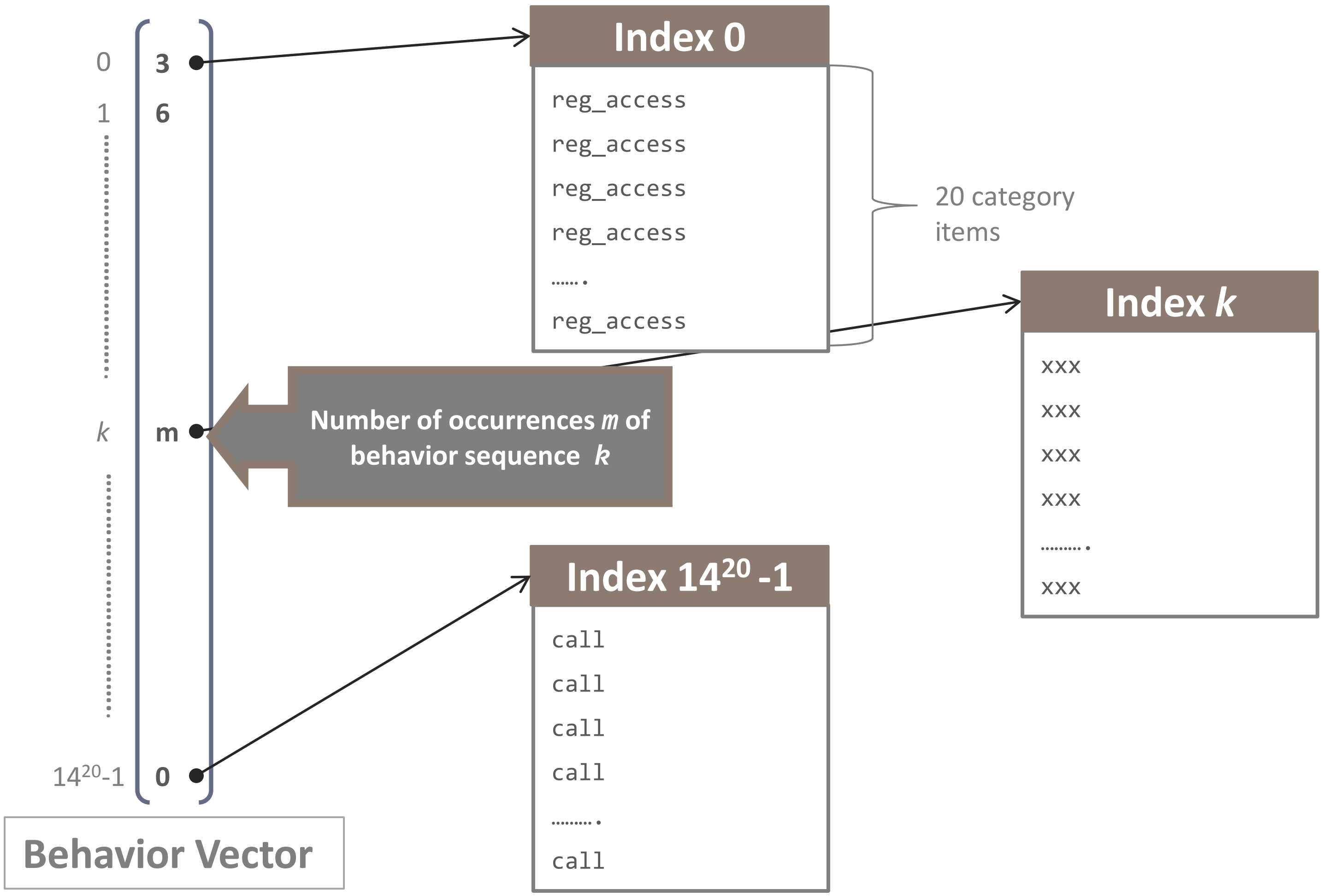
SEQ 6
reg_access
reg_access
reg_access

Binary Vector Generation

- Total possible behavior sequence permutations: ~ 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

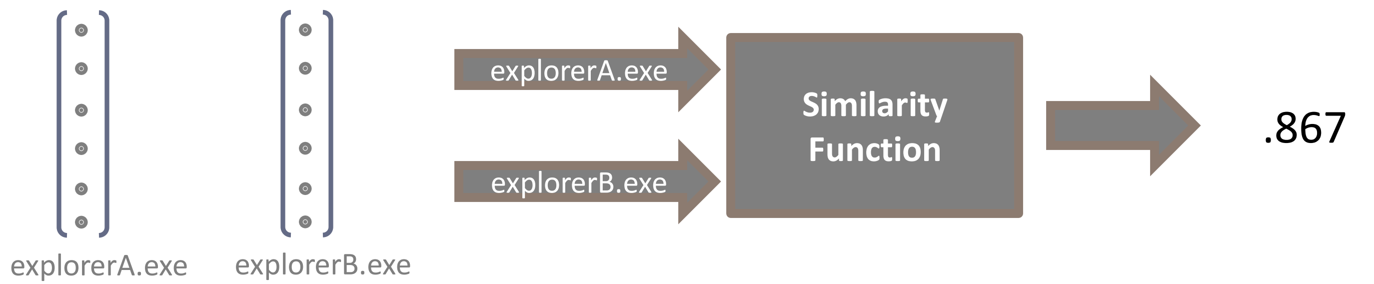
- **Naive Approach (continued…)**
  - Not very practical to implement directly
  - Fortunately, we can do better
- **Key observation**: Vector is sparse in that most of the dimensions will store the number '0'
- **Better Approach**
  - Only store the non-zero elements in memory
  - So if a binary has N total ir statements, then we only need to store at most N-`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:

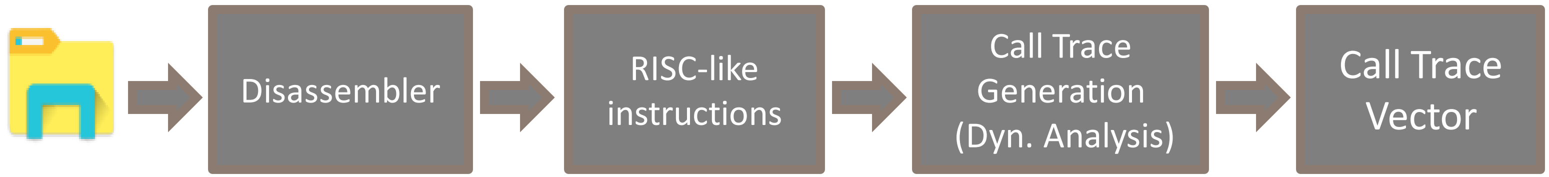
$$J(S_i, S_j) = \frac{S_i \cap S_j}{S_i \cup S_j} = \frac{\#\text{common behavior sequences}}{\#\text{total behaviors sequences}}$$

(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)	Call Trace (Code Path 2)	...	Call Trace (Code Path N)
main()	main()		main()
LoadLibraryW()	LoadLibraryW()		LoadLibraryW()
RegisterShellHook()	RegisterShellHook()		RegisterShellHook()
GetTokenInformation()	GetTokenInformation()		GetTokenInformation()
LsaOpenPolicy()	LsaOpenPolicy()		LsaOpenPolicy()
VirtualAlloc()	VirtualAlloc()		VirtualAlloc()
CreateEventW()	CreateEventW()		CreateEventW()
LogoffWindowsDialog()	LogoffWindowsDialog()		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 2
LoadLibraryW()
RegisterShellHook()
GetTokenInformation()

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

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()

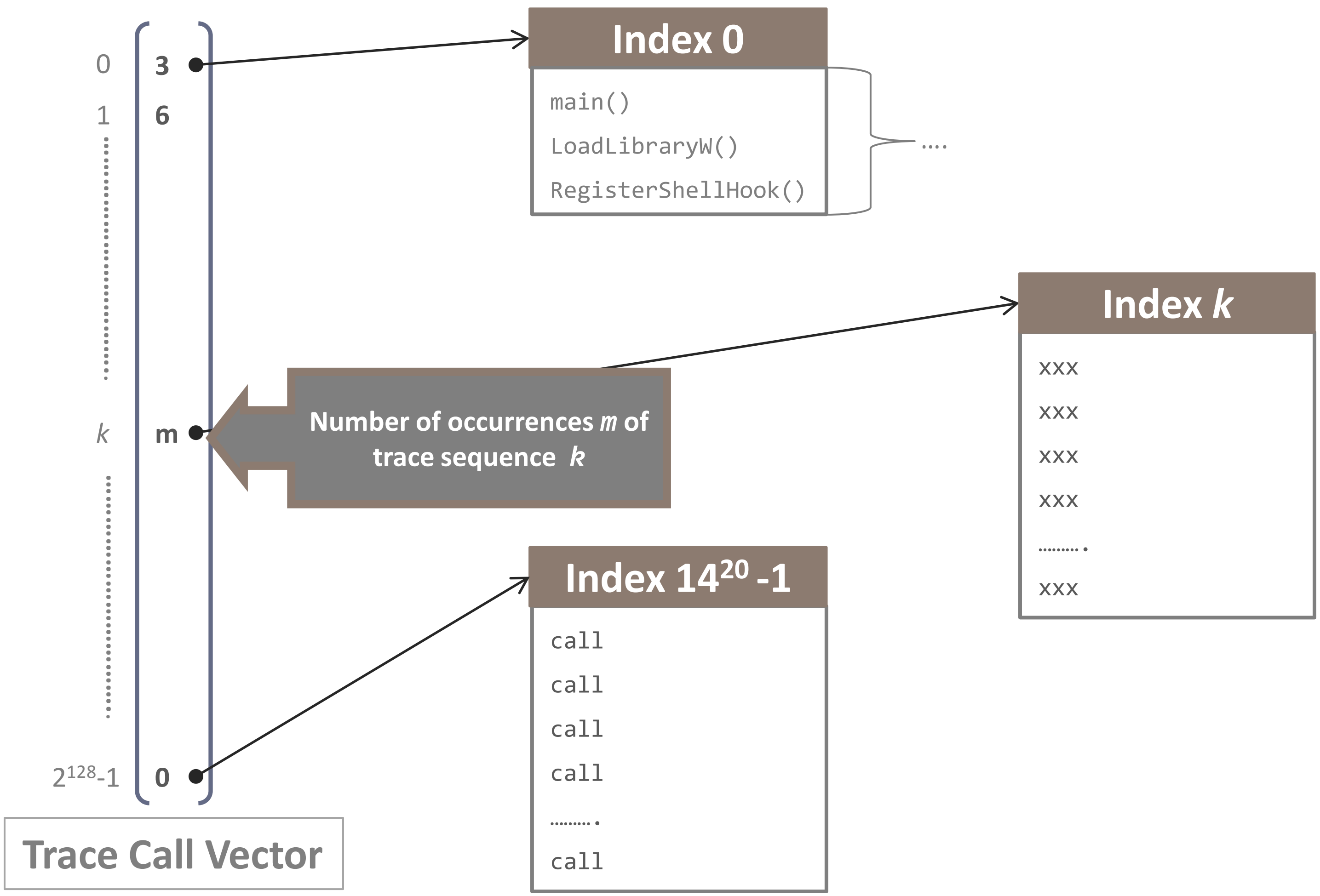
Trace Seq 2
LoadLibraryW()
RegisterShellHook()
GetTokenInformation()

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

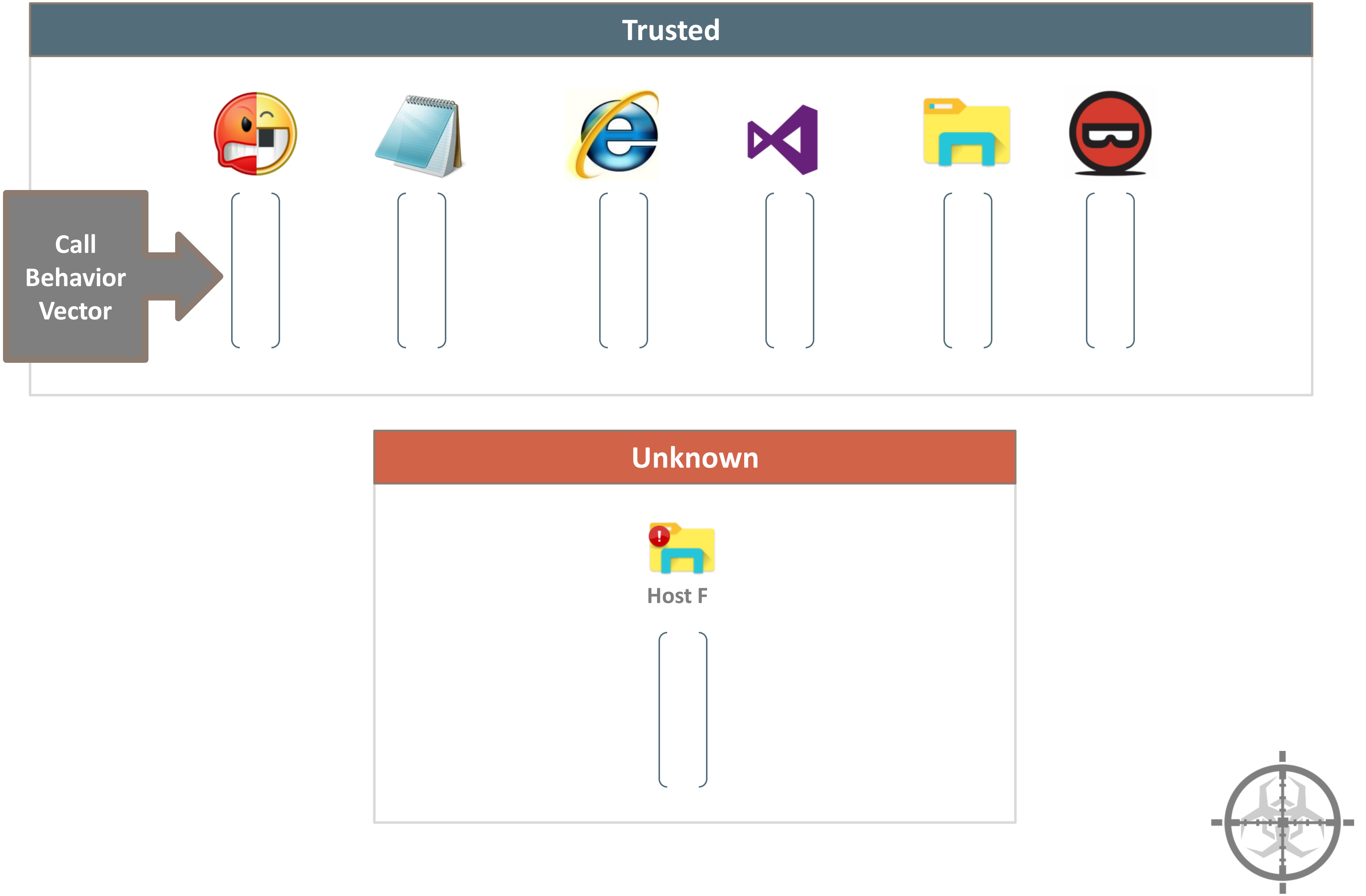
Trace Seq 4
GetTokenInformation()
LsaOpenPolicy()
VirtualAlloc()

Call Trace Vector Generation

- **Trace Call Vector**



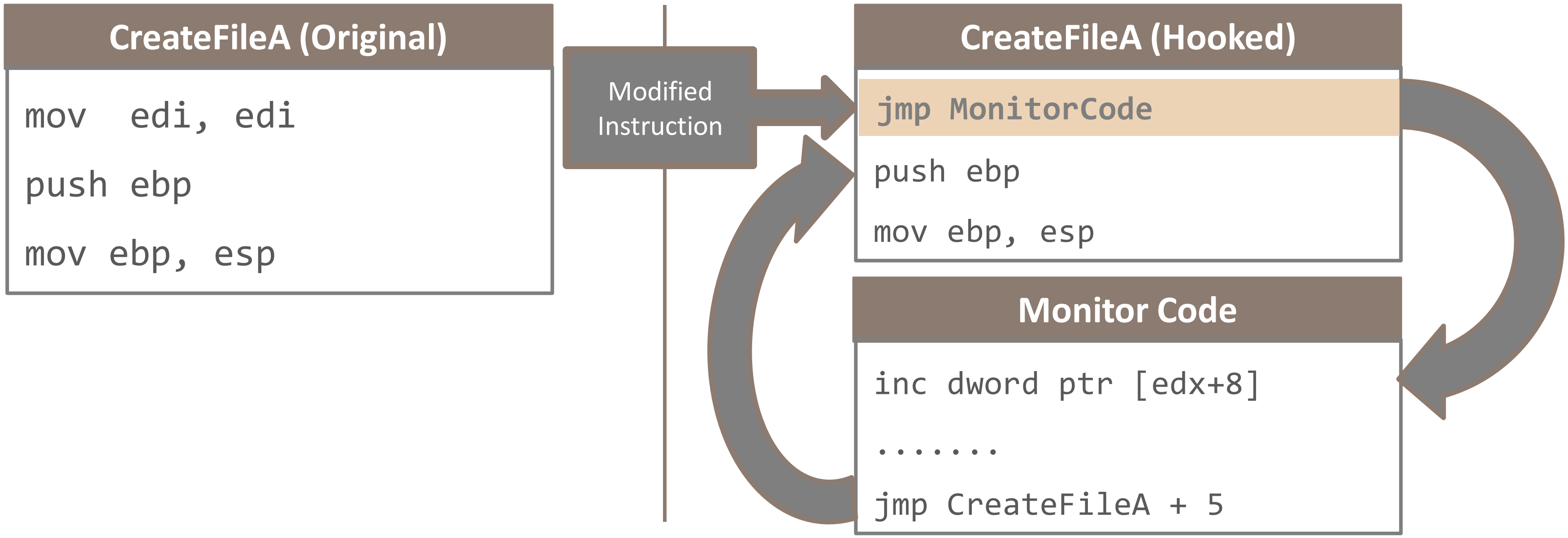
Call Trace Vector Generation



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***