

C.14 ALGORITMI AVANSATI FOLOSITI IN SECURITATEA CIBERNETICA

PAUL A. GAGNIUC



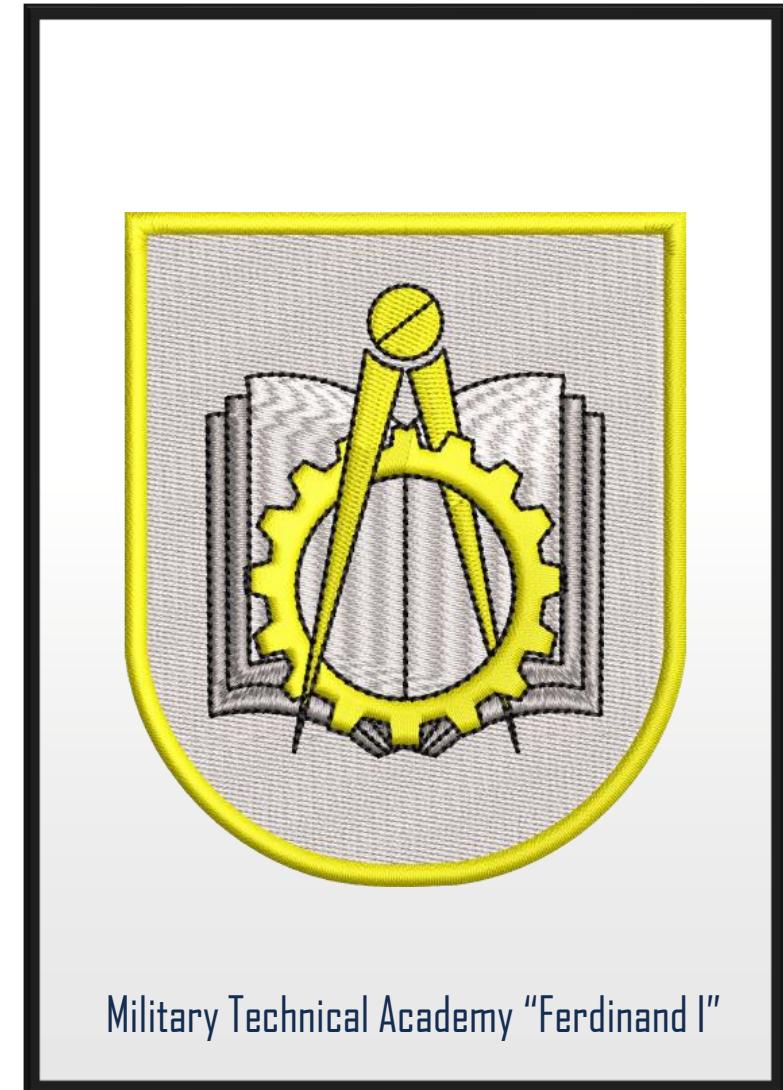
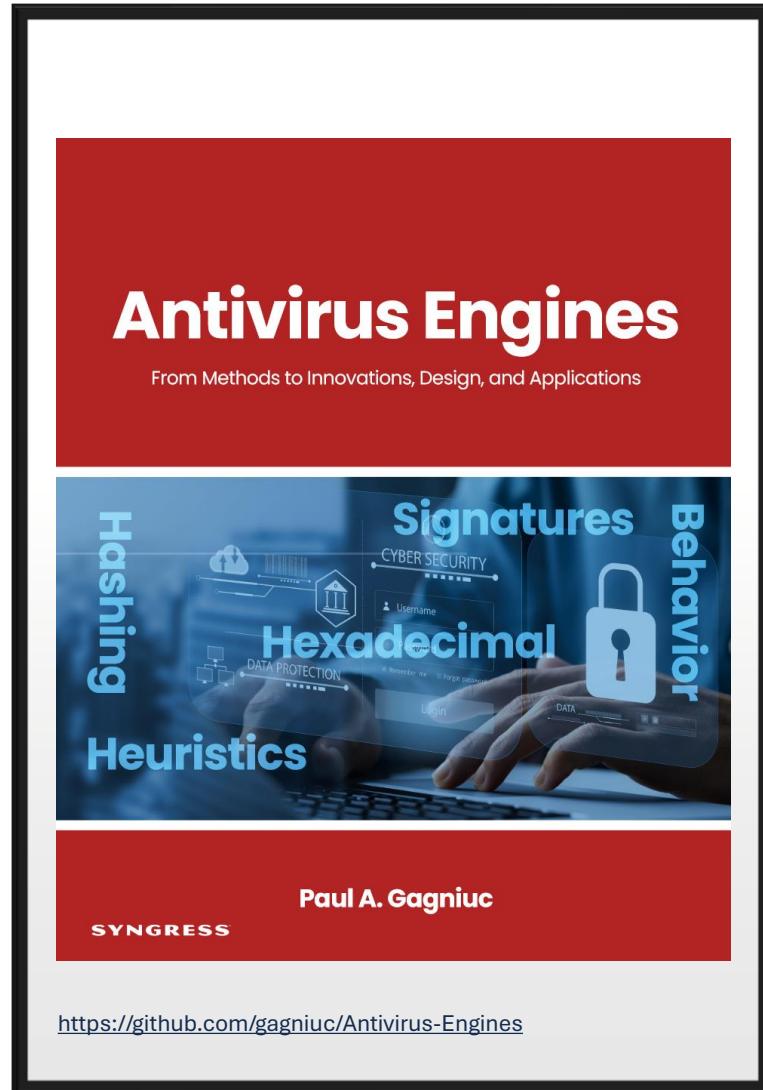
Academia Tehnică Militară „Ferdinand I”

PRINCIPALELE PĂRȚI ALE PREZENTĂRII

C.14 Algoritmi avansati folositi in securitatea cibernetica:

- **C.14.1 TIPURI DE SALTURI ȘI IMPLICAȚII PENTRU DETECTAREA MALWARE**
- **C.14.2 SEMNĂTURI MALWARE**
- **C.14.3 MOTOARE ANTIVIRUS**
- **C.14.4 ALGORITMI FOLOSITI IN SECURITATEA CIBERNETICA**

De citit:



De reținut:

- Antivirus ! Nu există unul „cel mai bun”, deoarece intervin parametri mulți și versiuni diferite.
- Antivirusurile se împart în două categorii: abordări îndrăznețe și abordări sigure (bazate pe prevenție).
- Ce ne dorim de la un antivirus? Consum redus de resurse; testele de tip benchmark sunt adesea irelevante.
- Tipuri de malware? Clasic și emergent.
- Metode de detecție: hashing, analiză hexadecimală, euristică.
- Pool-ul de semnături commune prezintă pericolul de overfitting.
- Pool-ul de exploituri a scăzut progresiv datorită restructurării sistemelor de operare.
- Se poate face update sistemului de operare, dar nu și utilizatorului.



Securitate cibernetica de la cercetare la profesor



C.14.1

TIPURI DE SALTURI ȘI IMPLICAȚII PENTRU DETECTAREA MALWARE



INSTRUCȚIUNEA E9 ESTE ABSOLUT NORMALĂ ÎN COD LEGITIM! CUM FOLOSEȘTE MALWARE-UL E9?

I. Hooking (de returnarea funcțiilor)

Malware-ul poate înlocui începutul unei funcții legitime (ex: CreateProcess, OpenFile) cu:

E9 <offset> ; jmp către codul său propriu

Astfel, când programul apelează funcția, sare în codul atacatorului

2. Redirecționarea fluxului de execuție

În virusi și troieni injectați în procese, apare des: E9 în secțiuni neașteptate (ex: .text modificat, sau .rdata corupt)

Redirecționează către cod ofuscat, loader, shellcode etc.

3. Obfuscare sau polimorfism

Malware-ul își „sperate” codul în bucăți și face salturi între ele cu jmp E9

Asta îngreunează analiza statică (ex: în IDA sau x64dbg)

E9 este opcode-ul pentru un salt lung

E9 xx xx xx xx → jmp <relativ offset>

Cum ne dăm seama dacă un E9 este suspect?

- Se află la începutul unei funcții standard? (ex: în kernel32.dll) → suspect
- Sare în afara modulului? → suspect
- Nu are referințe legitime (xref)? → suspect
- Este într-o secțiune marcată ca rdata sau idata? → foarte suspect

CUM FOLOSEŞTE MALWARE-UL E9?

HOOKING

```
format PE64 console
entry start

include 'win64a.inc'

section '.text' code readable executable

start:
    call hooked_function

    invoke ExitProcess, 0

hooked_function:
    jmp malicious_code ; E9 - simulare hook

legit_code:
    invoke MessageBoxA, 0, msg_legit, title, 0
    ret

malicious_code:
    invoke MessageBoxA, 0, msg_mal, title, 0
    ret

section '.data' data readable writeable

msg_legit db "Funcție legitimă", 0
msg_mal db "Funcție redirectionată (hooked)", 0
title db "Hooking E9", 0
```

Ce face acest exemplu?

Acest cod FASM simulează un **hook** aplicat unei funcții legitime.

- Funcția `hooked_function` ar trebui să apeleze `legit_code`
- Dar este înlocuită cu un **JMP (E9)** către `malicious_code`
- Astfel, controlul programului este deturnat spre un cod alternativ

Aceasta este o tehnică frecvent folosită de malware pentru a intercepta funcții din sistem sau din aplicații legitime.

 Apelul original este păstrat, dar deturnat!

CUM FOLOSEŞTE MALWARE-UL E9?

REDIRECTION

```
format PE64 console
entry start

include 'win64a.inc'

section '.text' code readable executable

start:
    jmp injected_code ; E9 în .text care sare în cod "injectat"

normal_execution:
    invoke MessageBoxA, 0, msg_normal, title, 0
    invoke ExitProcess, 0

Injected_code:
    invoke MessageBoxA, 0, msg_injected, title, 0
    jmp normal_execution

section '.data' data readable writeable

msg_normal db "Execuție normală", 0
msg_injected db "Cod injectat (simulat)", 0
title db "Redirecționare E9", 0
```

Ce ilustrează acest exemplu?

Acest cod simulează un caz de redirecționare a execuției folosind instrucțiunea jmp (E9), exact cum fac virușii și troienii injectați.

- Execuția ar trebui să înceapă normal
- Însă un jmp de la începutul programului duce în cod „injectat”
- Aceasta afișează un mesaj, apoi revine la fluxul original

Este o tehnică tipică de malware pentru a insera shellcode sau loaderi în secțiuni modificate precum .text sau .rdata.

CUM FOLOSEŞTE MALWARE-UL E9? OBFUSCATION

```
format PE64 console
entry start

include 'win64a.inc'

section '.text' code readable executable

start:
    jmp part1

part3:
    invoke MessageBoxA, 0, msg_final, title, 0
    invoke ExitProcess, 0

part1:
    xor rax, rax
    jmp part2

part2:
    add rax, 1
    jmp part3

section '.data' data readable writeable

msg_final db "Cod fragmentat prin E9", 0
title db "Obfuscare", 0
```

Ce face acest exemplu?

Acest cod demonstrează o tehnică simplă de **obfuscare** folosită de malware.

- Codul este „spart” în bucăți (part1, part2, part3)
- Execuția sare între ele folosind jmp (E9)
- Deși logic ar putea fi scris secvențial, e fragmentat intenționat

Această tehnică este folosită pentru a îngreuna analiza statică în decompilatoare precum IDA sau Ghidra, unde codul pare întrerupt și greu de urmărit.



MALWARE OPCODE SIGNATURES:

Malware_Opcode_Signatures.yar

```

rule Hooking_JMP_E9
{
    strings:
        $hook = { E9 ?? ?? ?? ?? ?? }
    condition:
        $hook
}

rule NOP_Sled
{
    strings:
        $sled = { 90 90 90 90 90 }
    condition:
        $sled
}

rule Jump_Obfuscation
{
    strings:
        $jmp = { EB ?? EB ?? EB ?? }
    condition:
        $jmp
}

rule Shellcode_Allocator
{
    strings:
        $alloc = { 6A 00 68 ?? ?? ?? ?? E8 }
    condition:
        $alloc
}

rule Reflective_Jump
{
    strings:
        $vjmp = { FF 15 ?? ?? ?? ?? E9 }
    condition:
        $vjmp
}

```

Acstea sunt **semnături binare** care detectează **pattern-uri de instrucțiuni (opcode-uri)** caracteristice pentru comportamente folosite frecvent de malware.

Acste semnături pot fi:
 ▲ mai generale (cu mai mulți ? → tolerant la variații)
 ▲ mai specifice (cu expresii regulate în stringuri complexe)
 ▲ bazate pe expresii incluzând și adrese, registre

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
• PS C:\Users\Paul> & C:/Users/Paul/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Paul/Desktop/New folder/scan.py"
  ▲ Găsit cod suspect în: c:/Users/Paul/Desktop/New folder/exemplu.exe
    - Regula: NOP_Sled
      ↴ Match brut: $sled
○ PS C:\Users\Paul>

```

Semnături comune de malware folosind wildcards (??):

1. Hooking – E9 ?? ?? ?? ?? ??
2. NOP sled – 90 90 90 90 90
3. Obfuscation jump – EB ?? EB ?? EB ??
4. Shellcode alloc – 6A 00 68 ?? ?? ?? ?? E8
5. Reflective loader (VirtualAlloc + JMP) – FF 15 ?? ?? ?? ?? E9

Acste semnături sunt euristice – ele caută modele comune în malware, nu secvențe exacte de fișiere.

```

import yara
import os

# Cale relativă:
current_dir = os.path.dirname(os.path.abspath(__file__))
rules = yara.compile(filepath=os.path.join(current_dir,
"Malware_Opcode_Signatures.yar"))
target_file = os.path.join(current_dir, "exemplu.exe")

matches = rules.match(target_file)

if matches:
    print(f"Găsit cod suspect în: {target_file}")
    for match in matches:
        print(f"- Regula: {match.rule}")
        for s in match.strings:
            print(f"  ↴ Match brut: {repr(s)}")
else:
    print(f"Nimic suspect în: {target_file}")

```

`pip install yara-python`

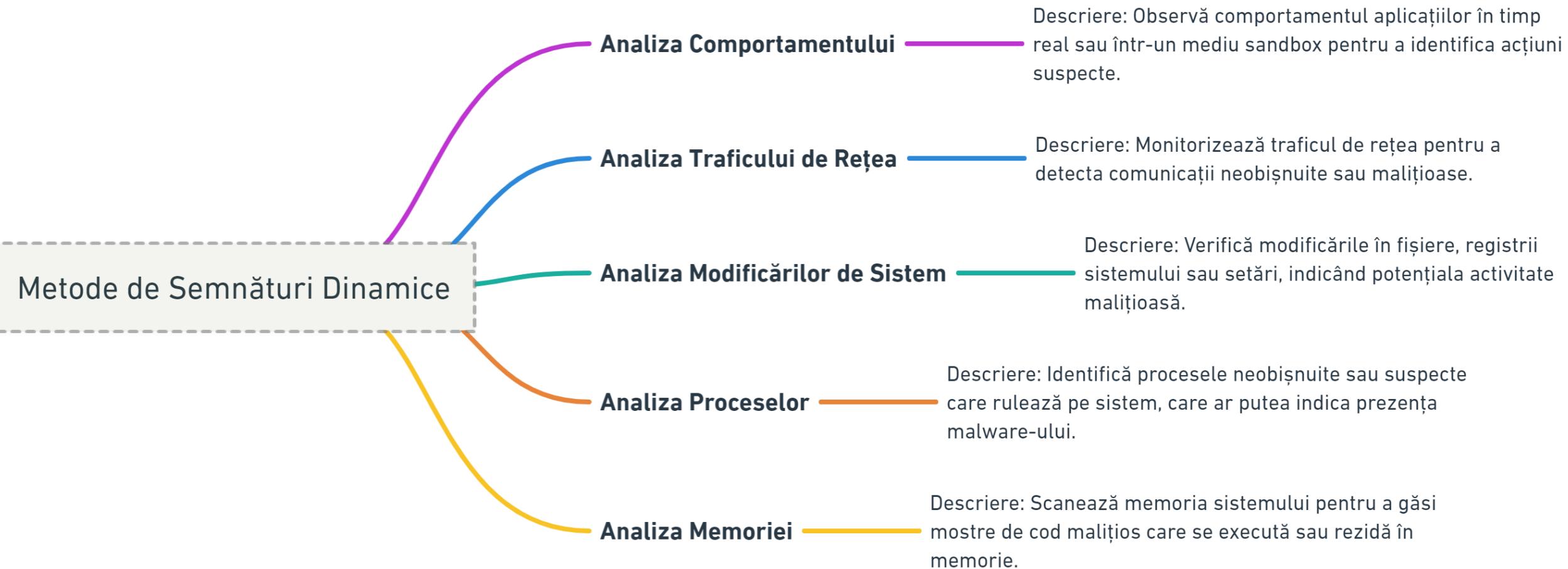
c.10.2

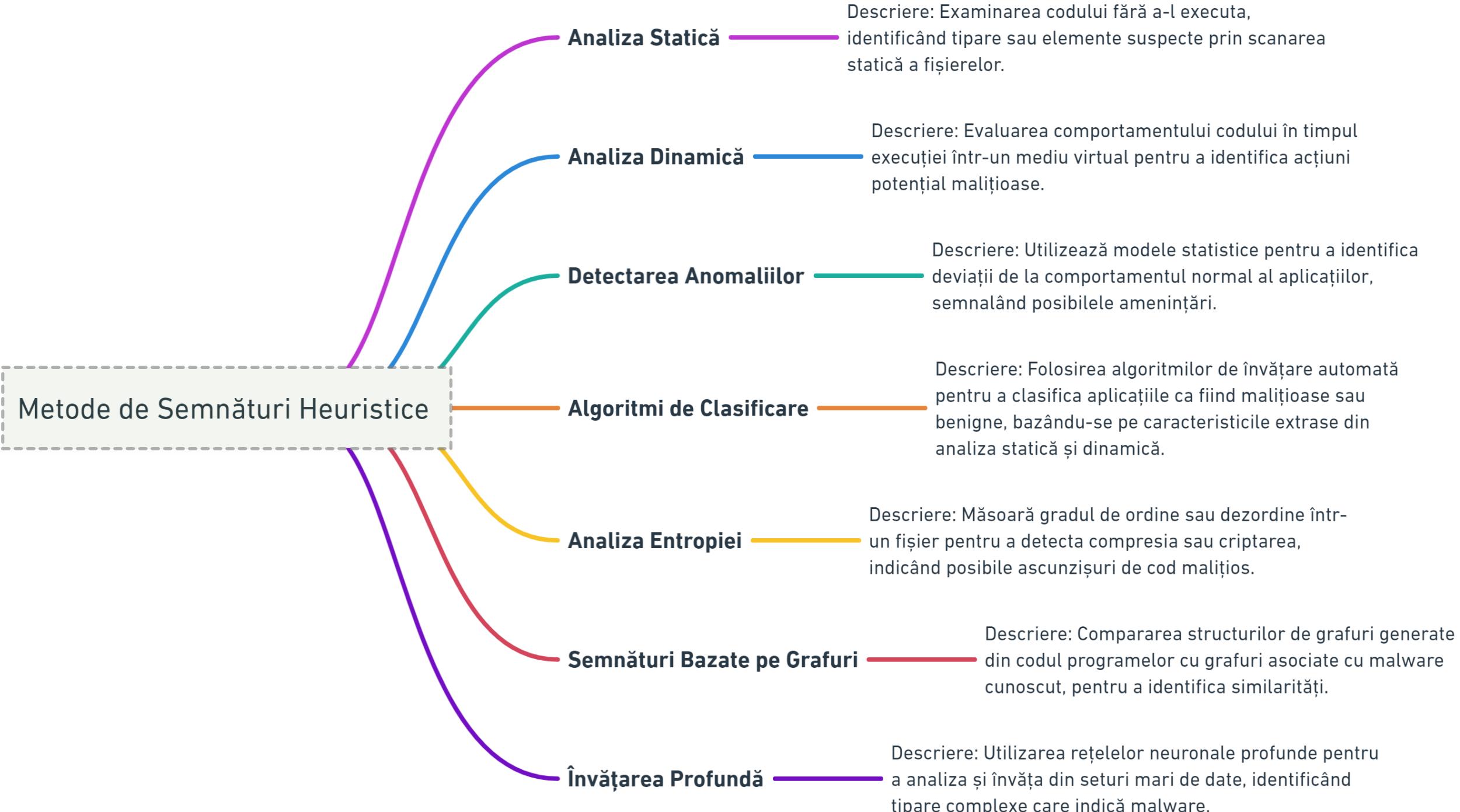
SEMNĂTURI MALWARE

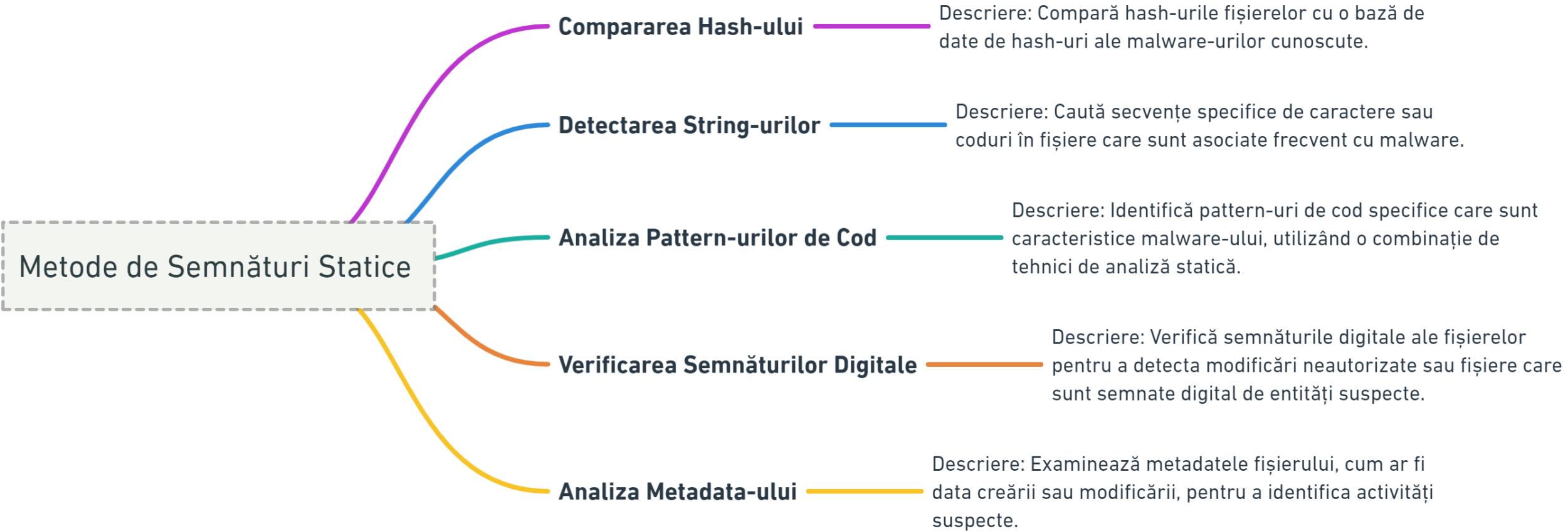


Metode de Semnături de Malware

- Semnături Statice** — Descriere: Identifică malware-ul prin compararea hash-urilor fișierelor suspecte cu o bază de date de hash-uri ale malware-urilor cunoscute.
- Semnături Dinamice** — Descriere: Se bazează pe analiza comportamentului la executarea într-un mediu controlat (sandbox) pentru a genera semnături bazate pe acțiunile detectate.
- Semnături Bazate pe String-uri** — Descriere: Caută secvențe specifice de text sau cod într-un fișier care sunt caracteristice malware-ului.
- Semnături Heuristiche** — Descriere: Folosește reguli sau algoritmi pentru a identifica tipare sau caracteristici ale codului care sugerează un comportament malitios.
- Semnături de Polimorfism** — Descriere: Detectează variantele de malware care folosesc tehnici de polimorfism pentru a-și modifica codul la fiecare infecție, îngreunând detectarea.
- Semnături de Anomalie** — Descriere: Identifică activitățile suspecte sau neobișnuite bazându-se pe un profil de normalitate stabilit anterior.



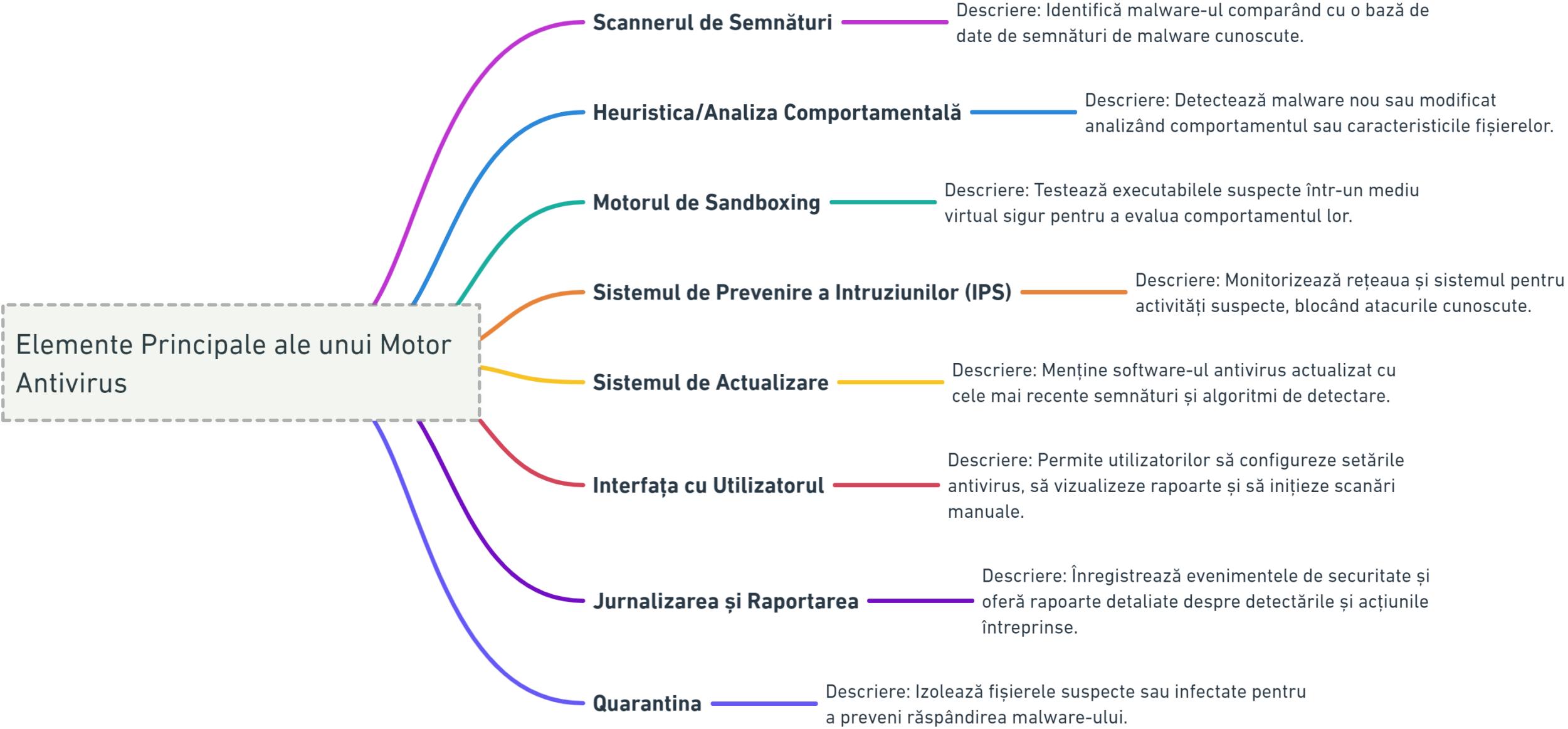


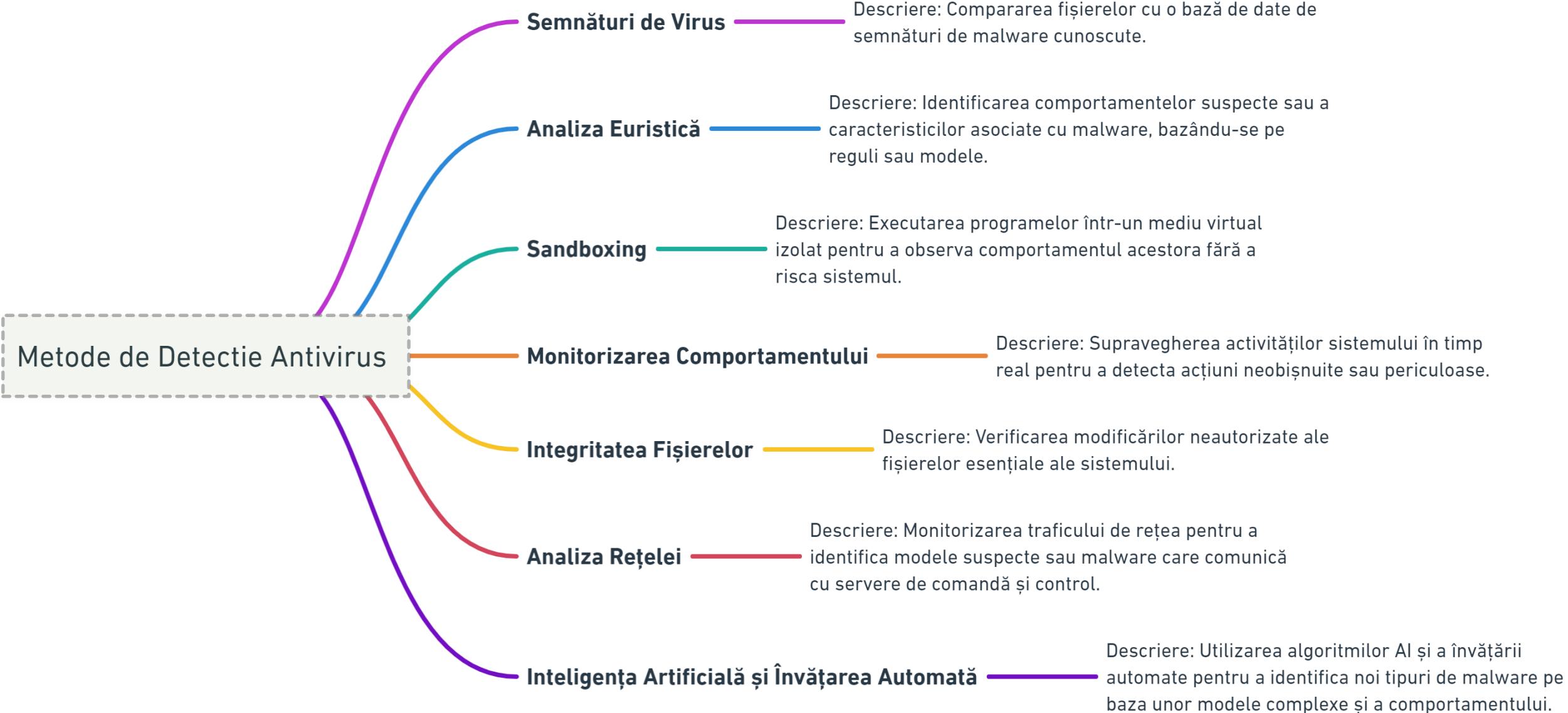


C.10.3

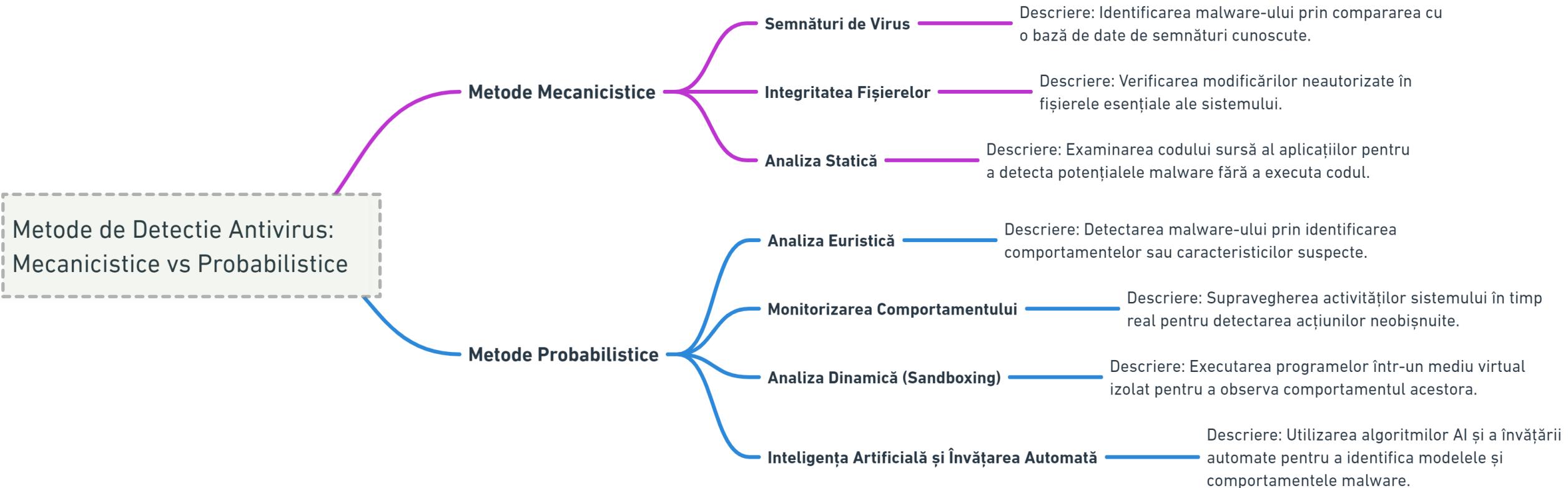
MOTOARE ANTIVIRUS

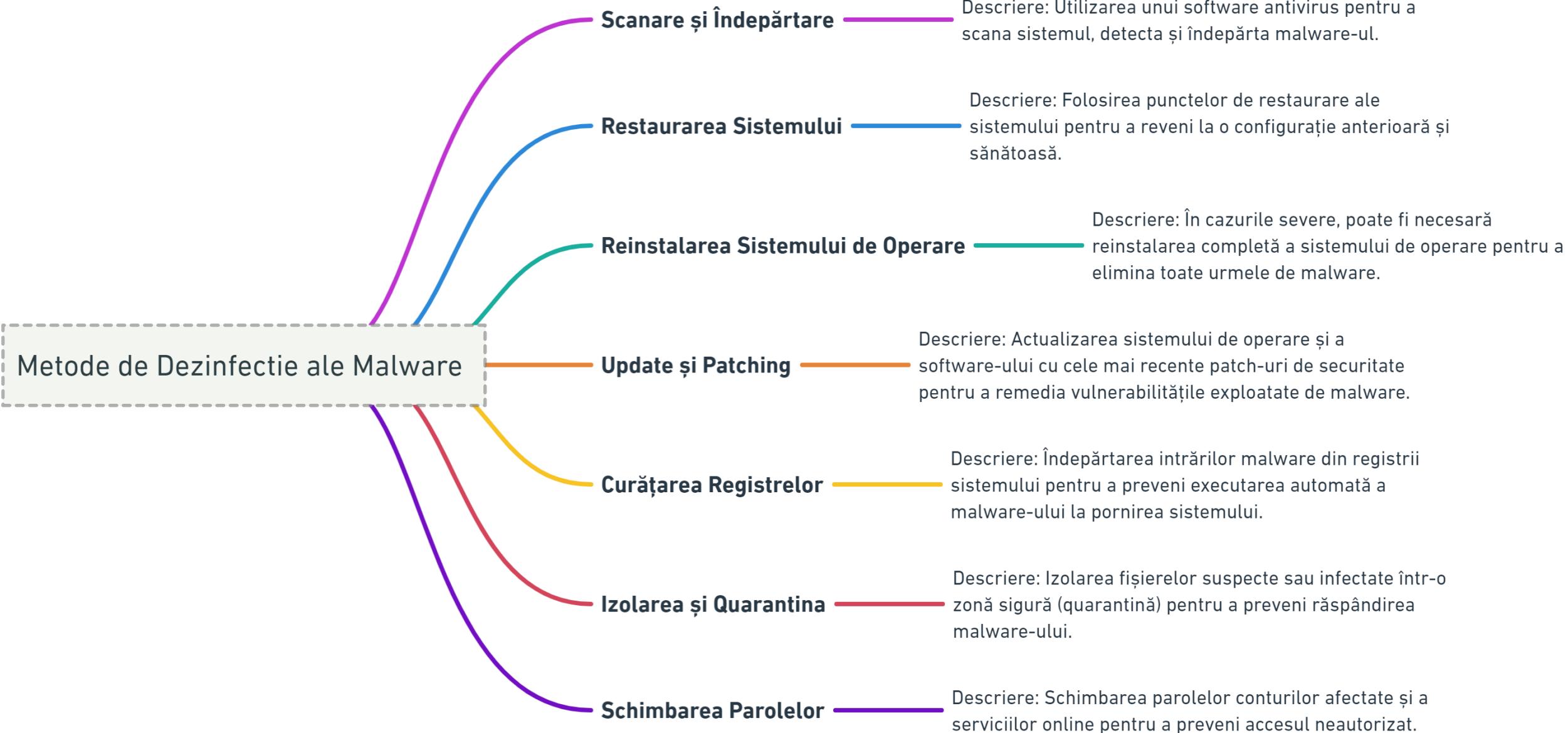


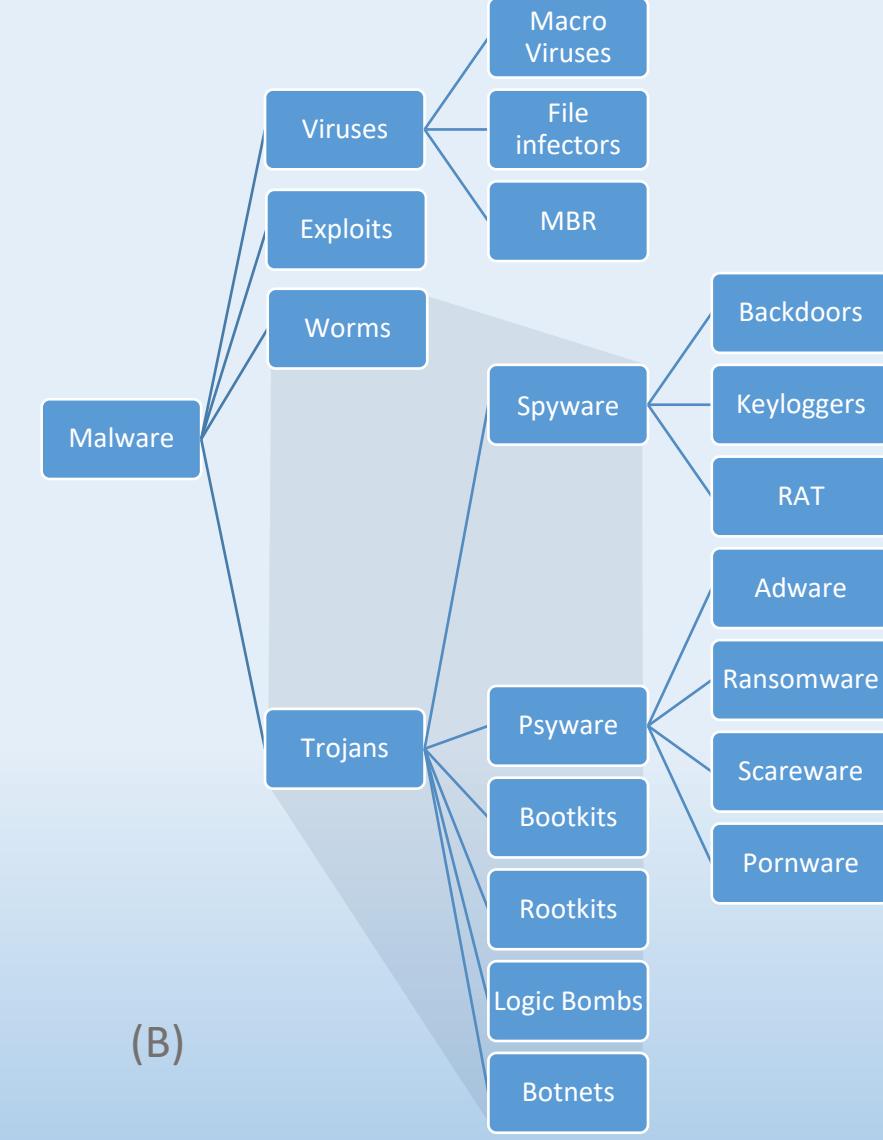
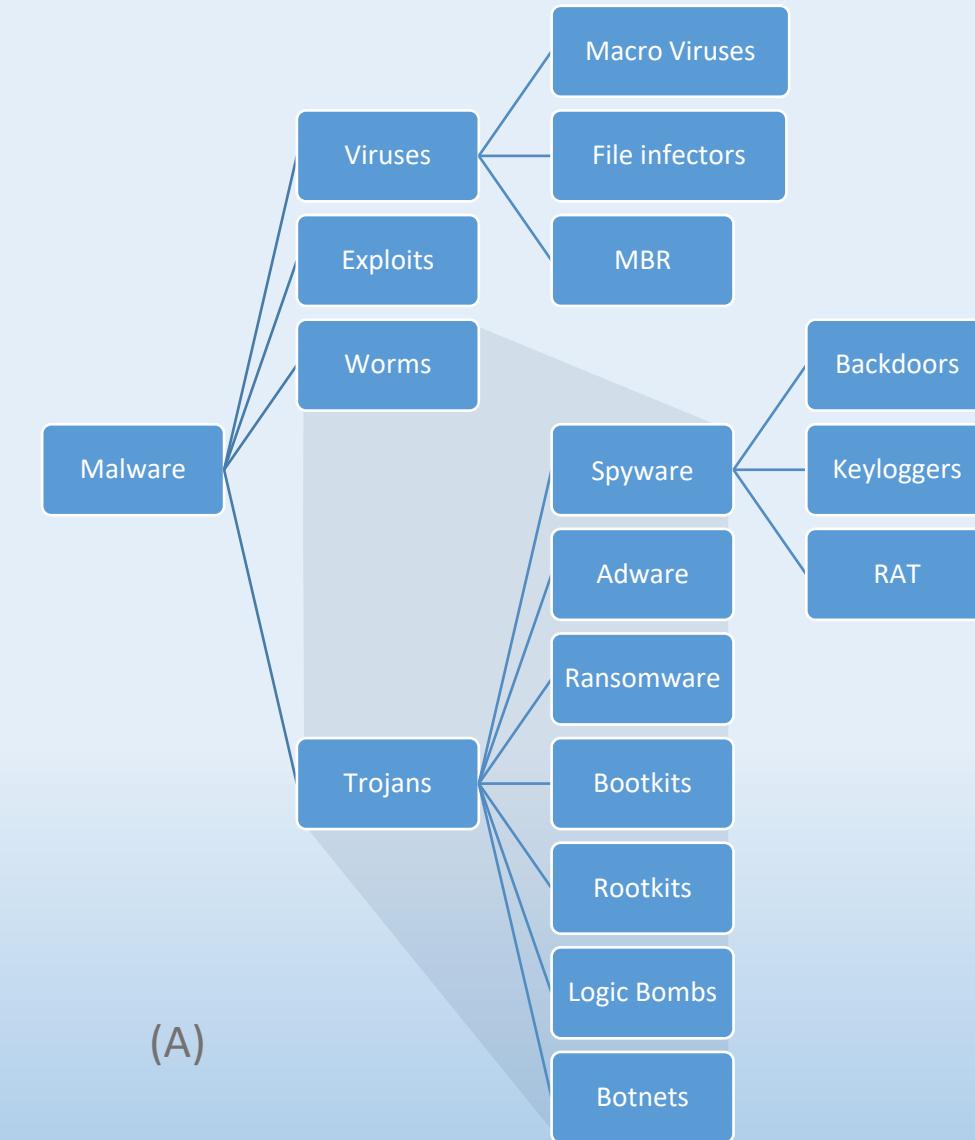




Noțiuni elementare: detecție mecanicistă vs detecție probabilistică.







non-polymorphic

Region Match

MD5 hash
signatures

db

Hash : 345853ea3c6d7d5947be300d3cfe2e5b
Hex (I) : E8 23 A9 42 C1 F0 10
Hex (II) : E8 23 ?? ?? C1 F0 ?? 77

Chunks Match

Hexadecimal
signatures

db

Semnături vs cod polimorf

Frequency Match

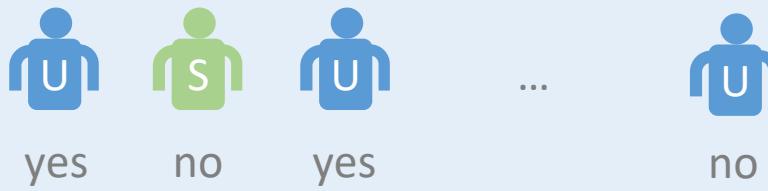
Heuristic
signatures

db

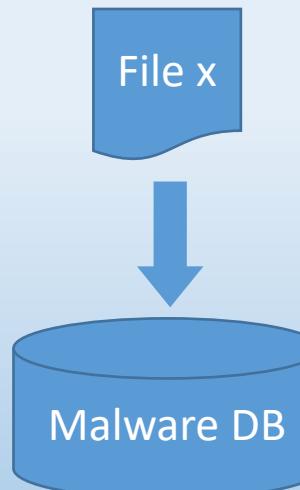
polymorphic

Catalogarea după părerea utilizatorilor:

Is this malware?



$$f(x) = \sum_{i=1}^m (\text{weight}(u_i) \times \text{vote}(x))$$

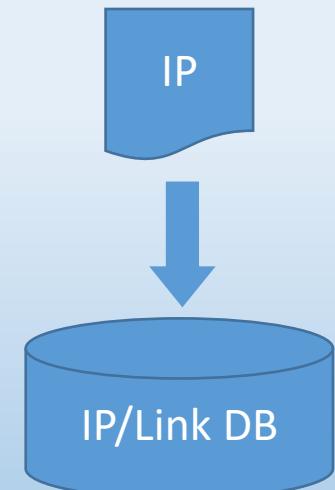


(A)

Is this IP/link malicious?

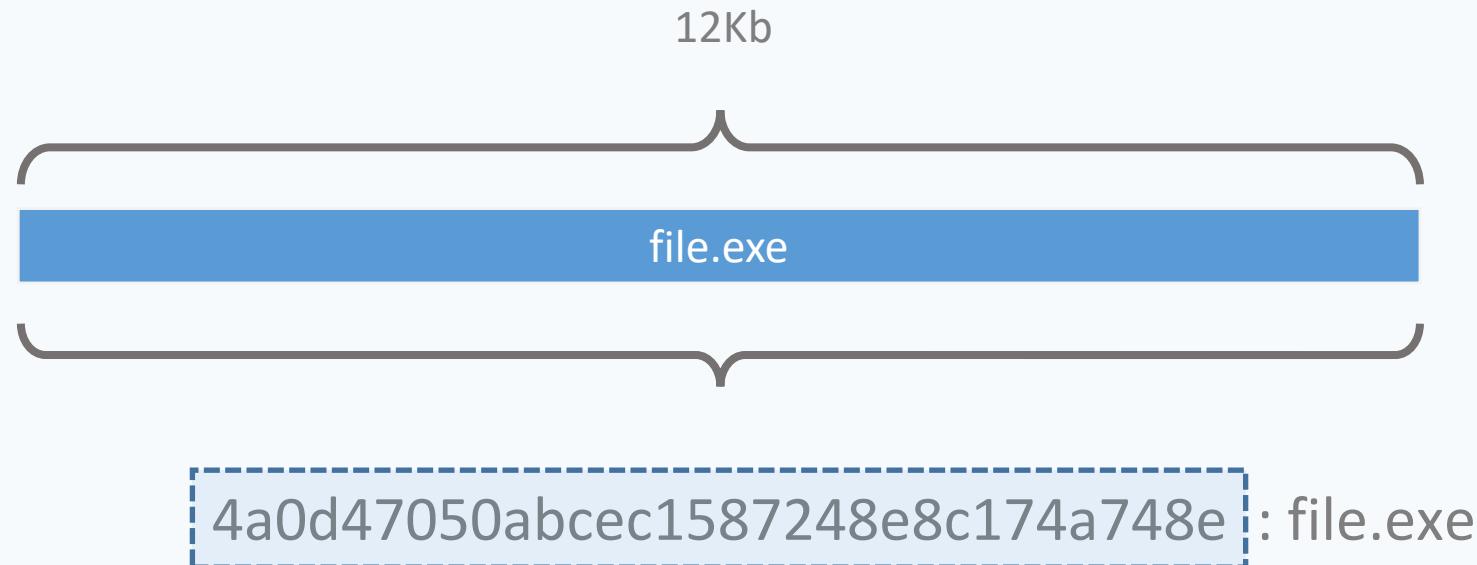


$$f(x) = \sum_{i=1}^m (\text{weight}(u_i) \times \text{vote}(x))$$

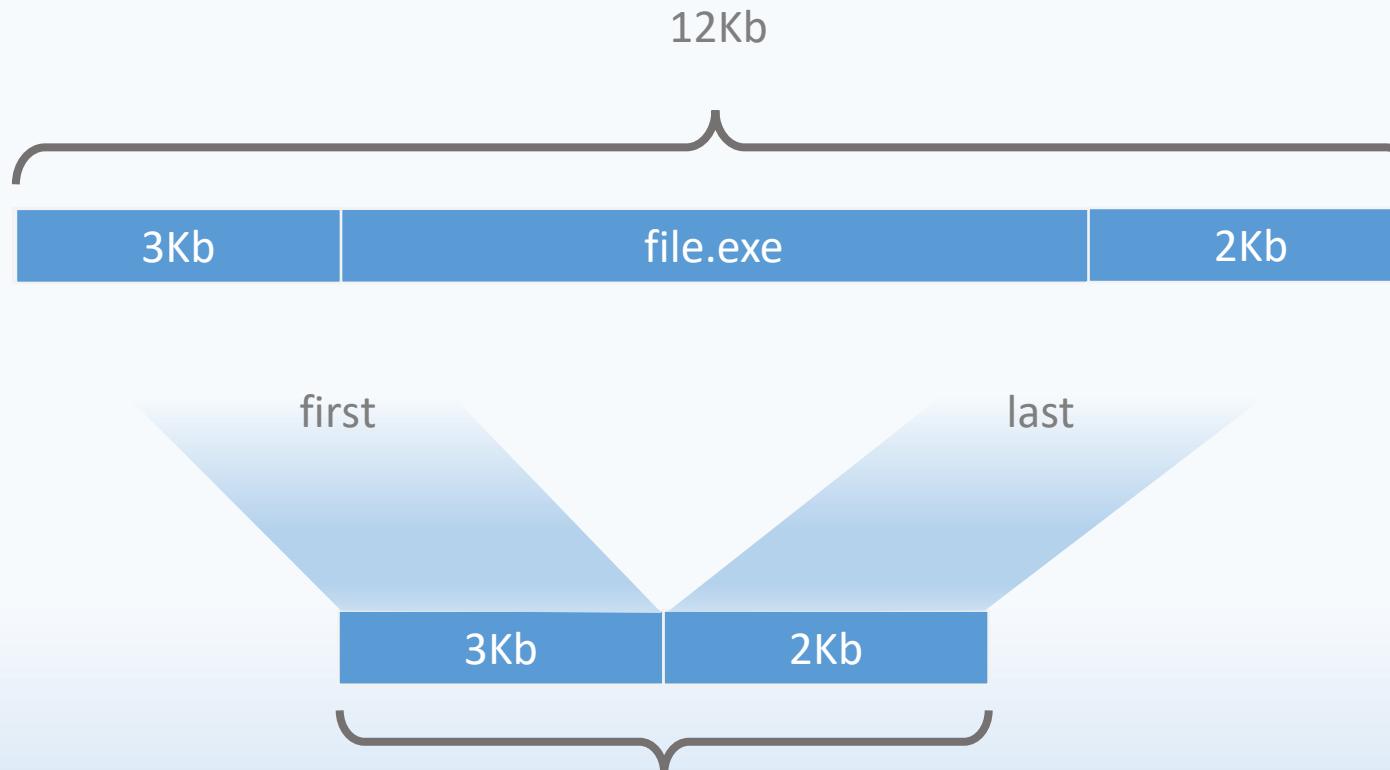


(B)

MD5 signature

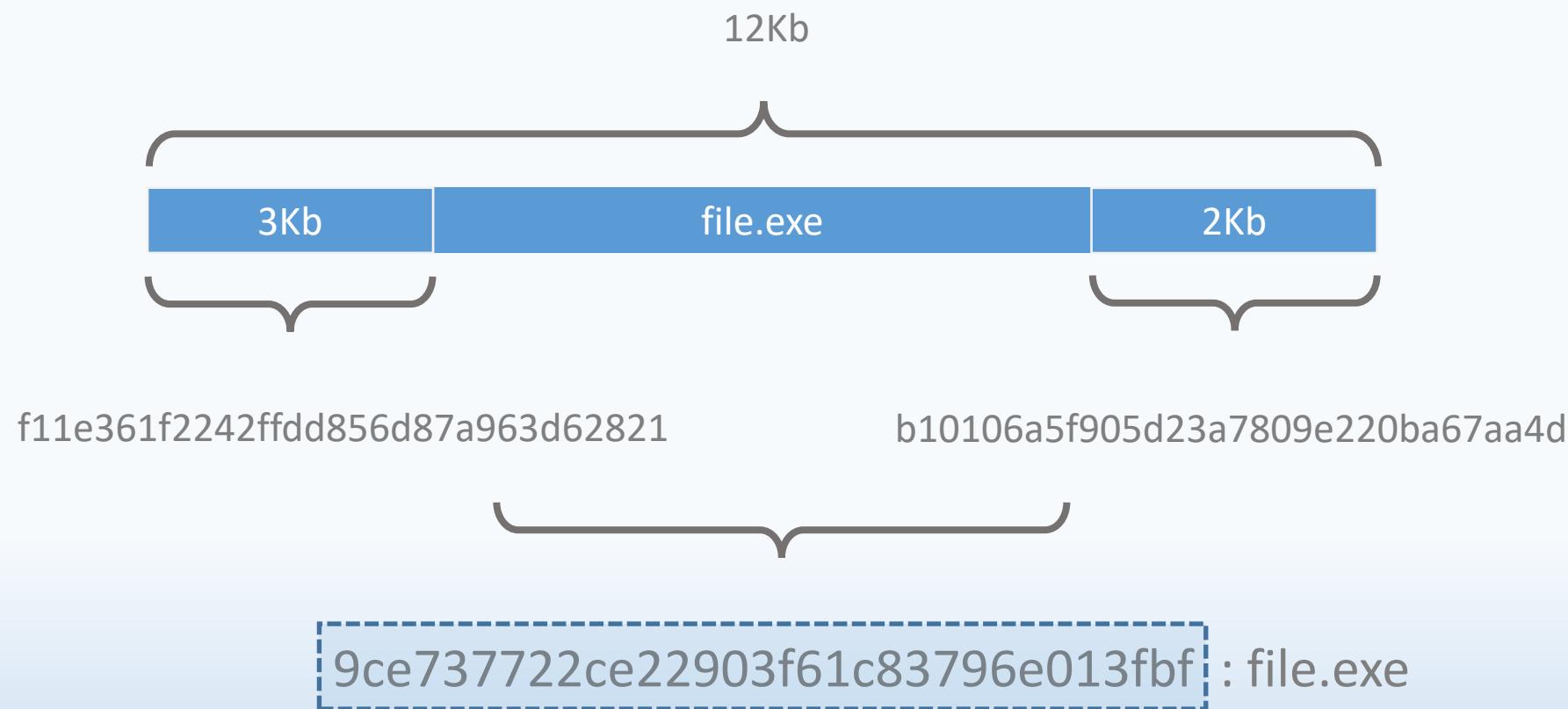


MD5 signature

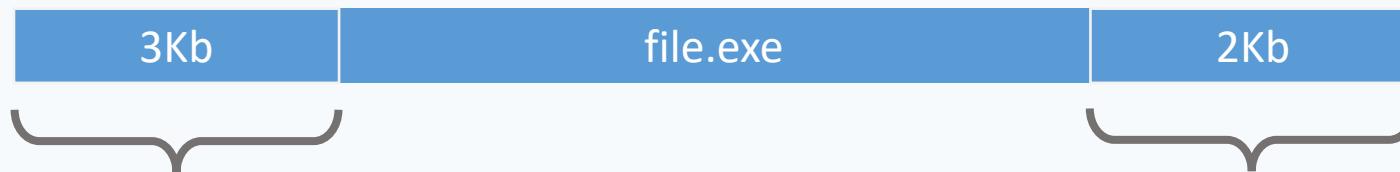


931ea08b1f3c1df63413b128f4c46f66 : file.exe

MD5 signature



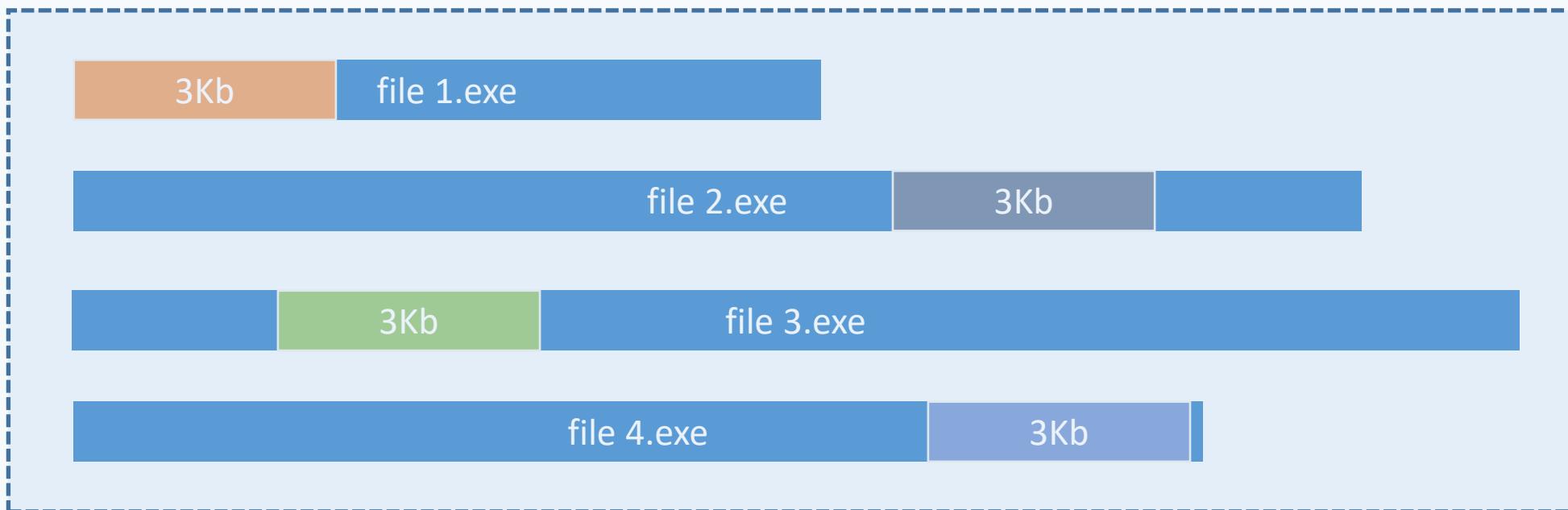
MD5 signature



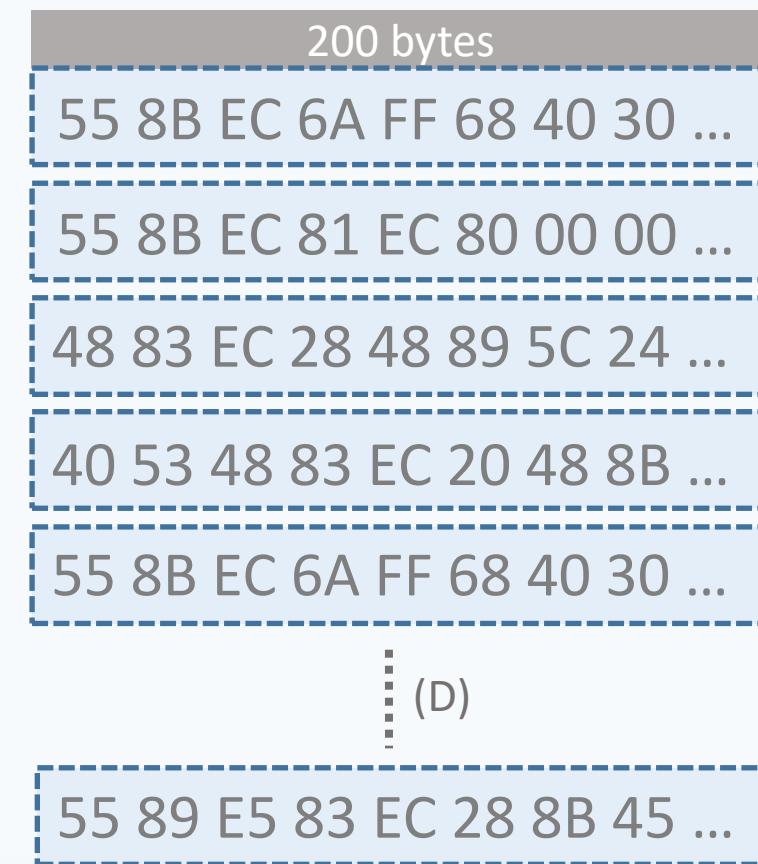
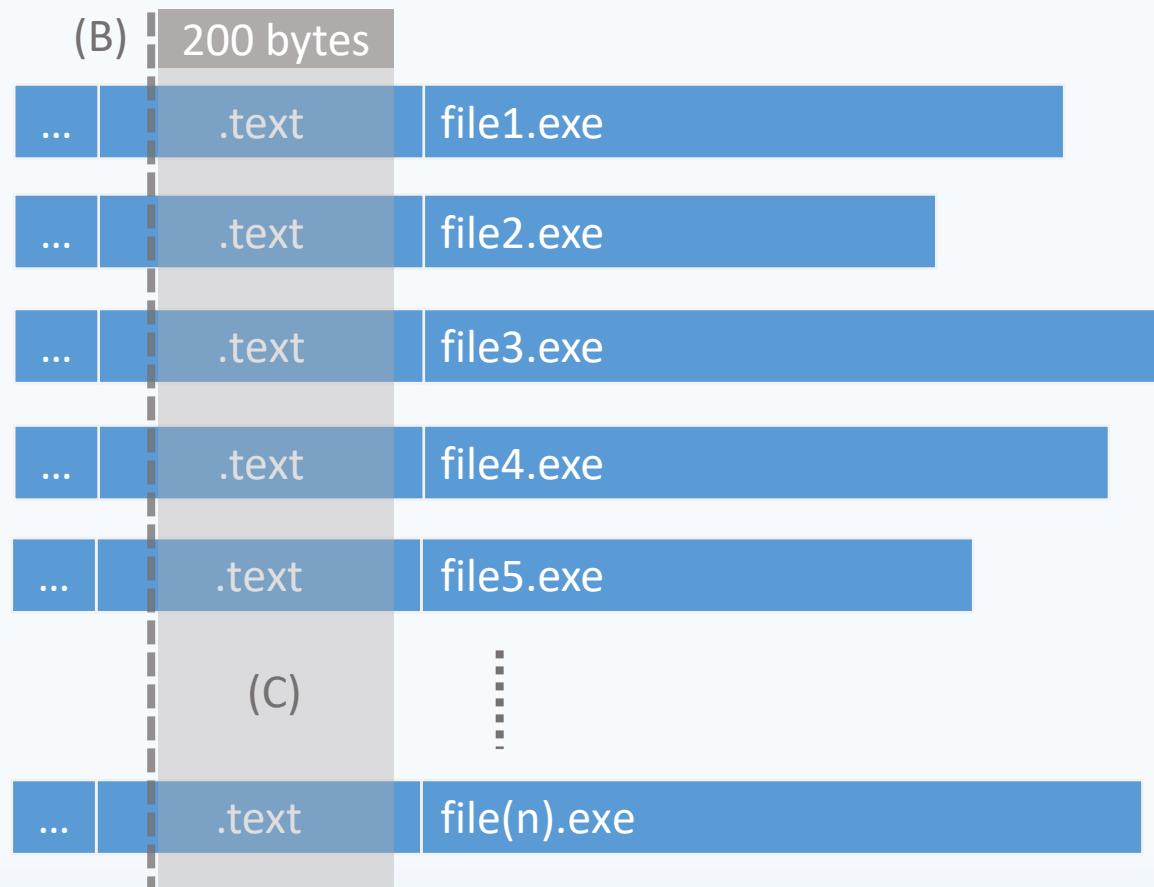
931ea08b1f3c1df63413b128f4c46f66

9ce737722ce22903f61c83796e013fbf

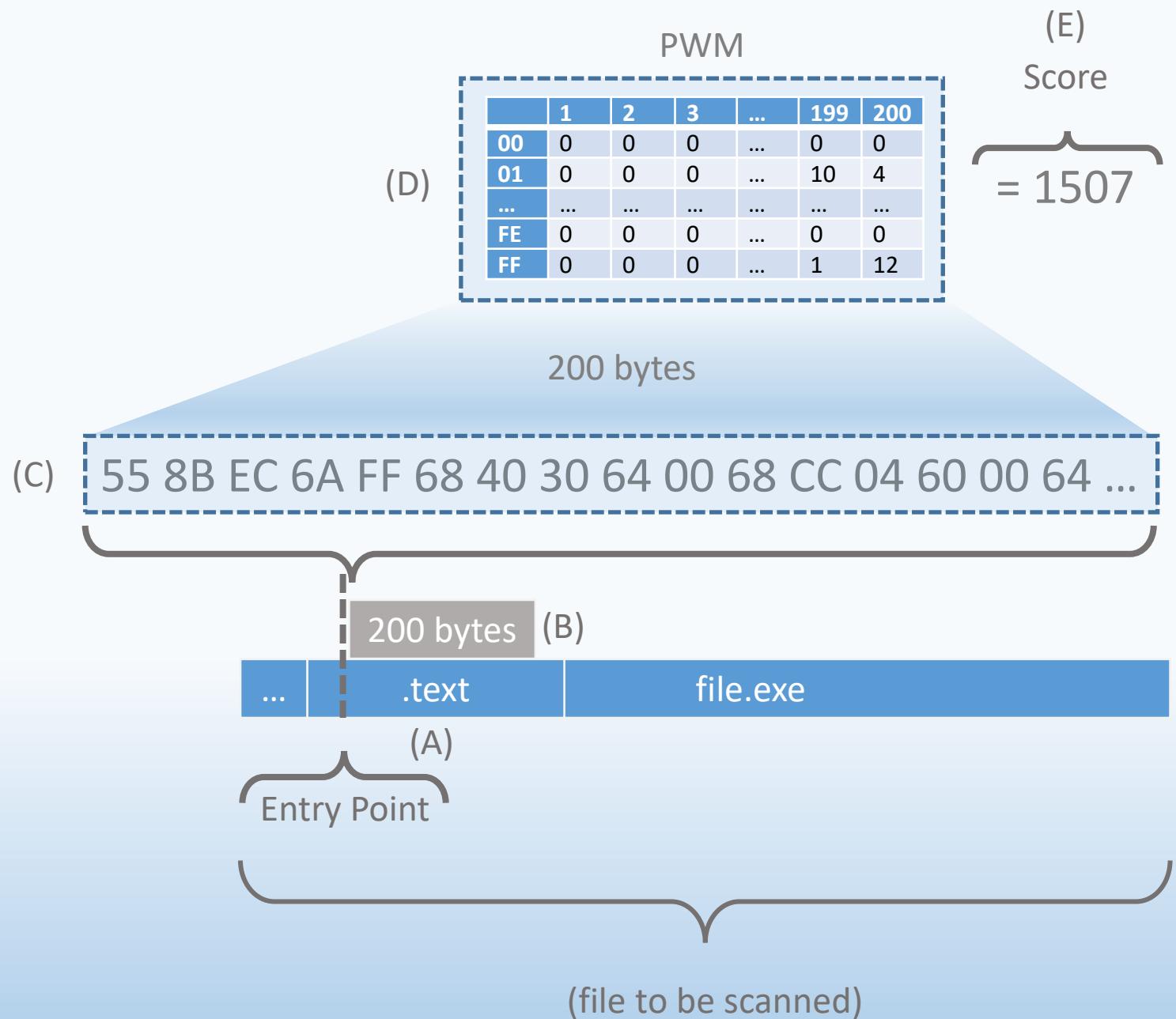
7fa24b9db9b313523977d14b27a7a676 : 9ce737722ce22903f61c83796e013fbf : file.exe

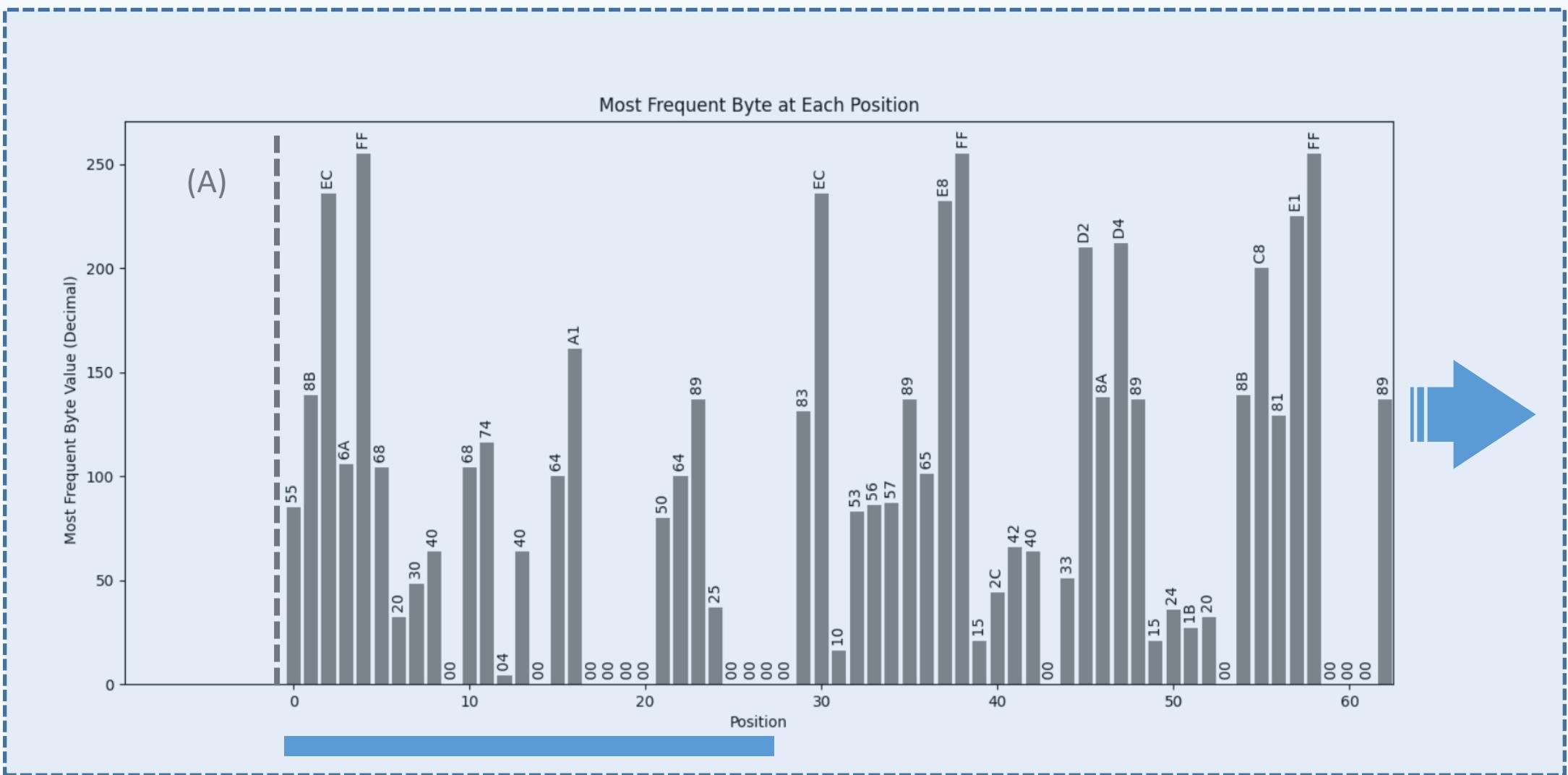


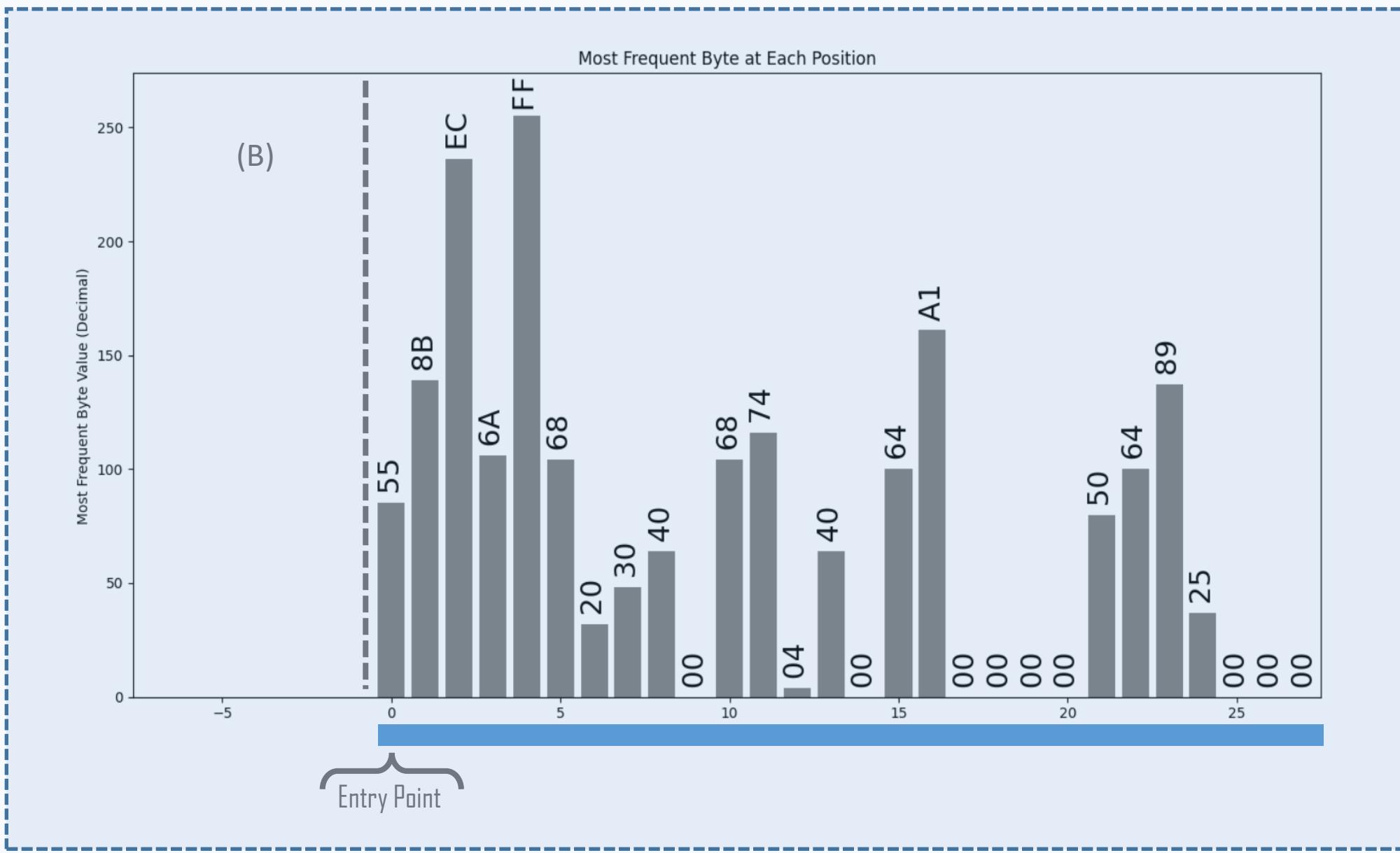
aa06d4dd7af1a739ecdce4630a2f7ed9

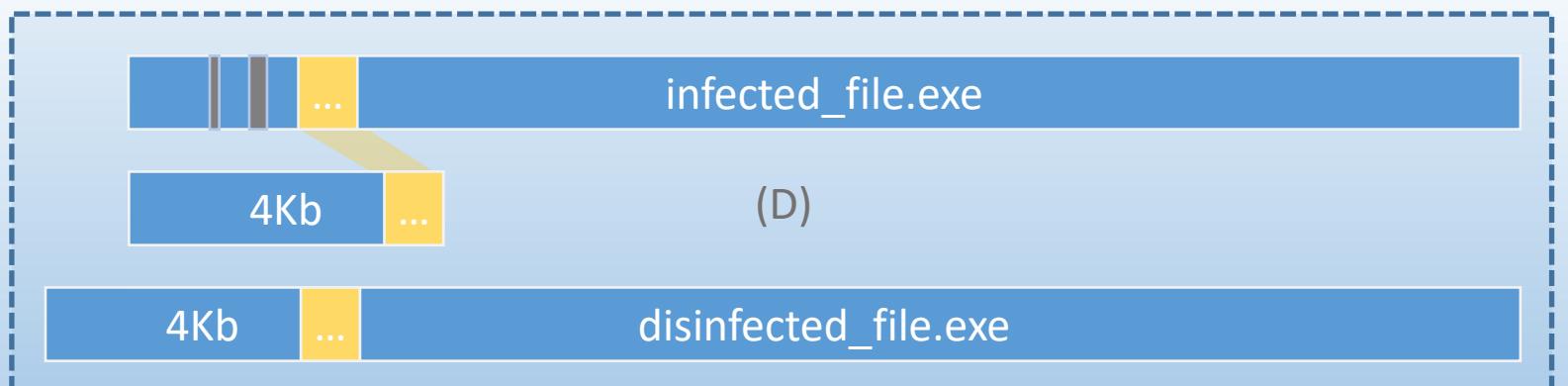
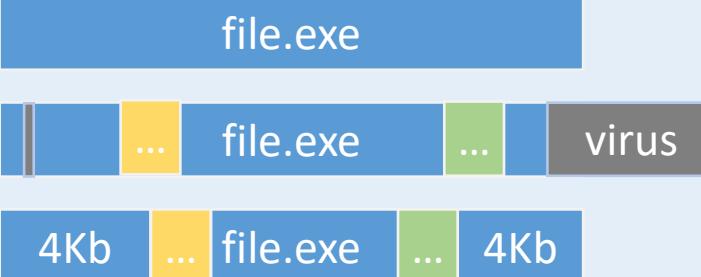
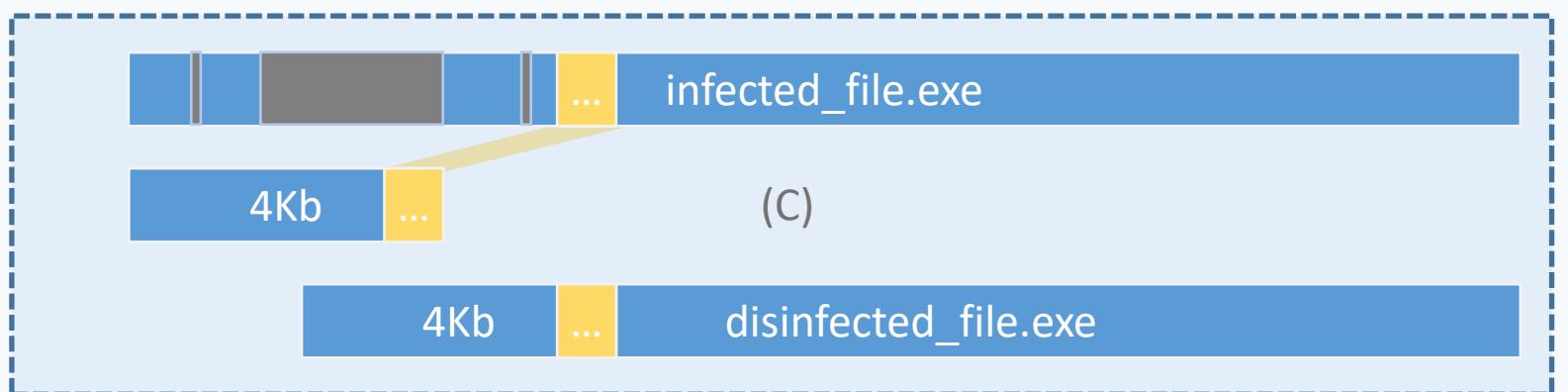
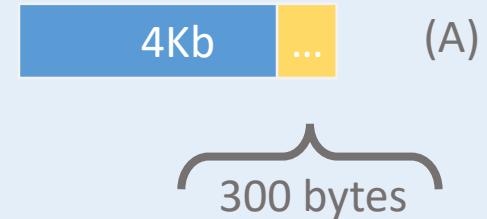
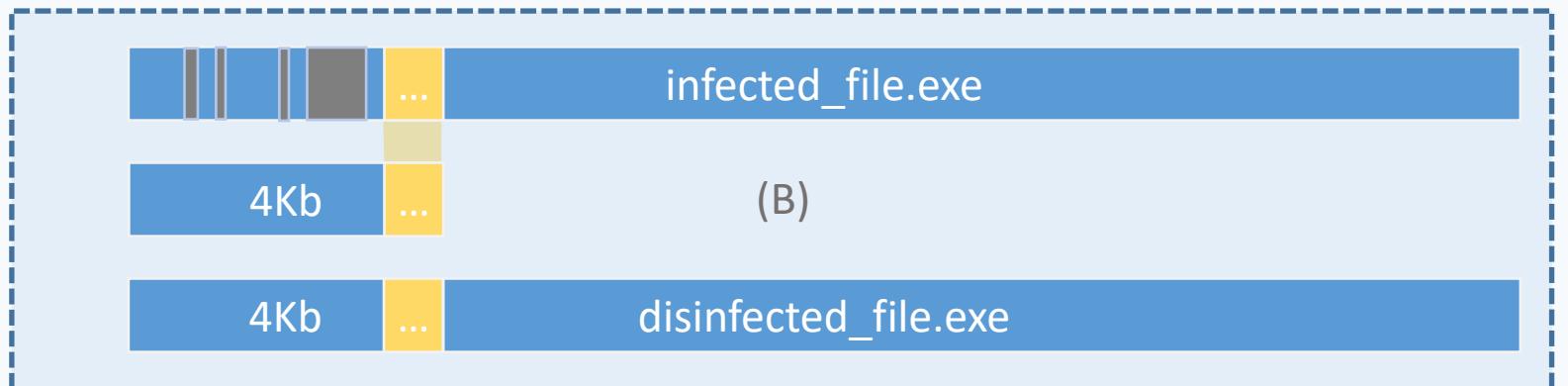


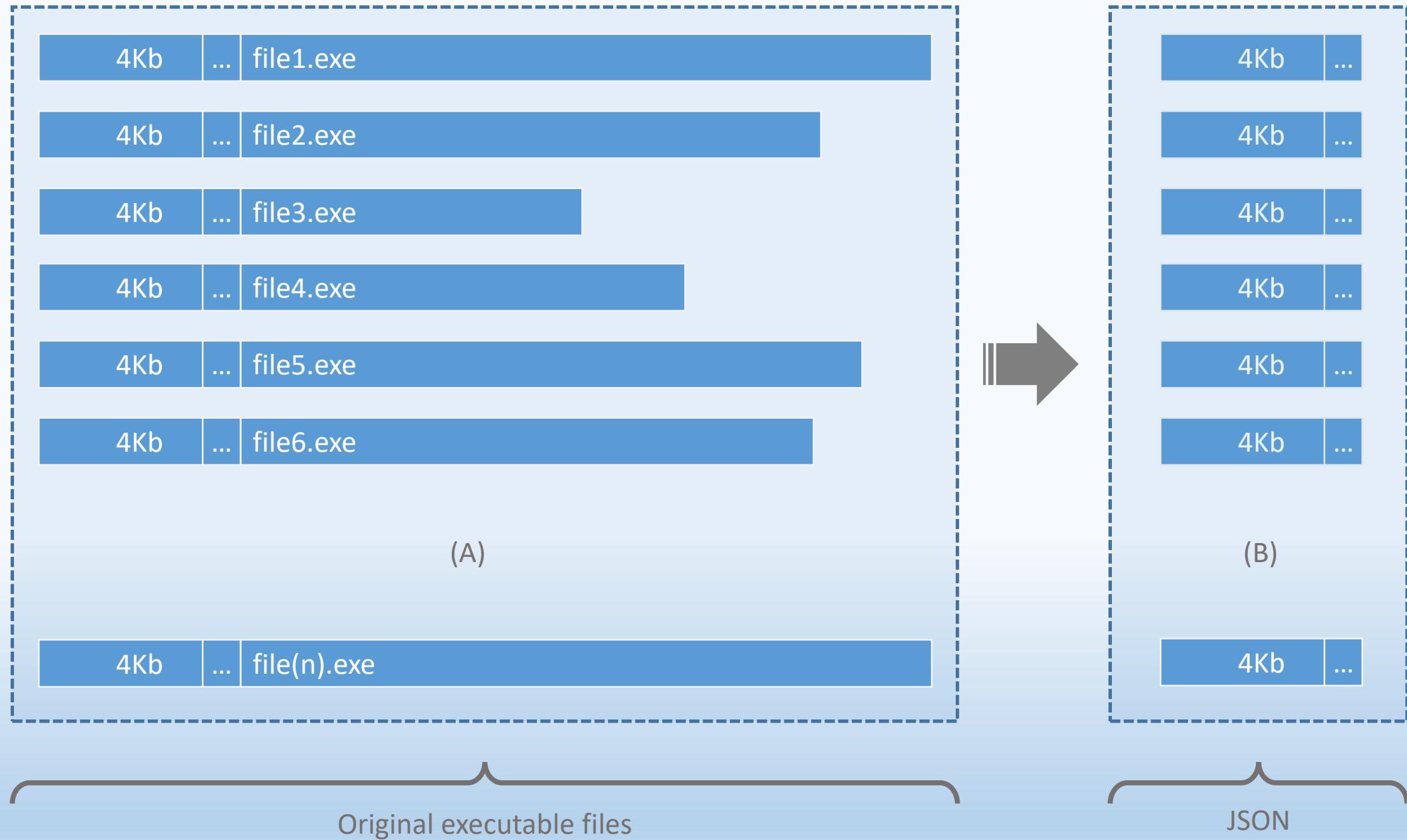
	1	2	3	...	199	200
00	0	0	0	...	0	0
01	0	0	0	...	10	4
...
FE	0	0	0	...	0	0
FF	0	0	0	...	1	12

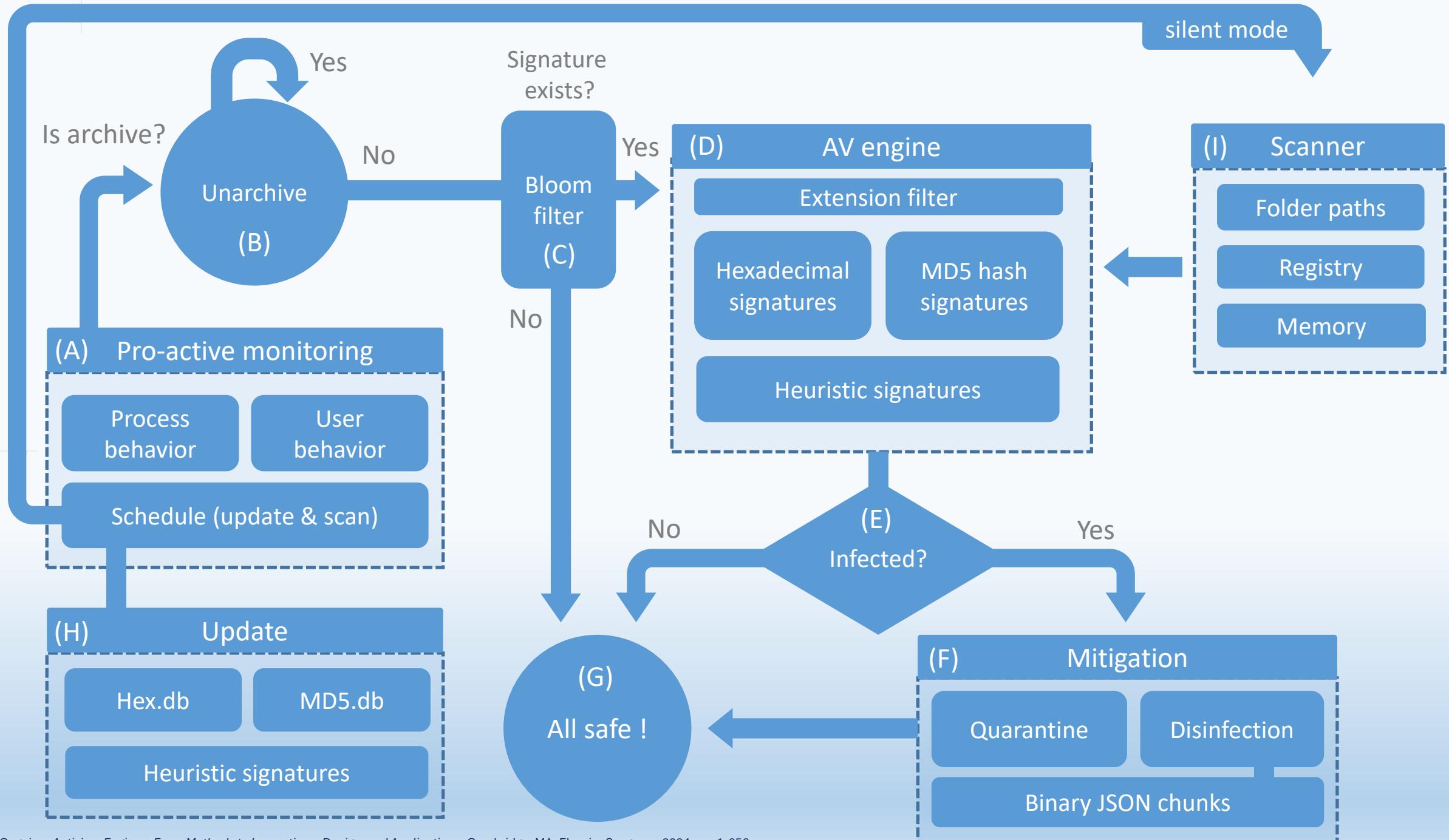


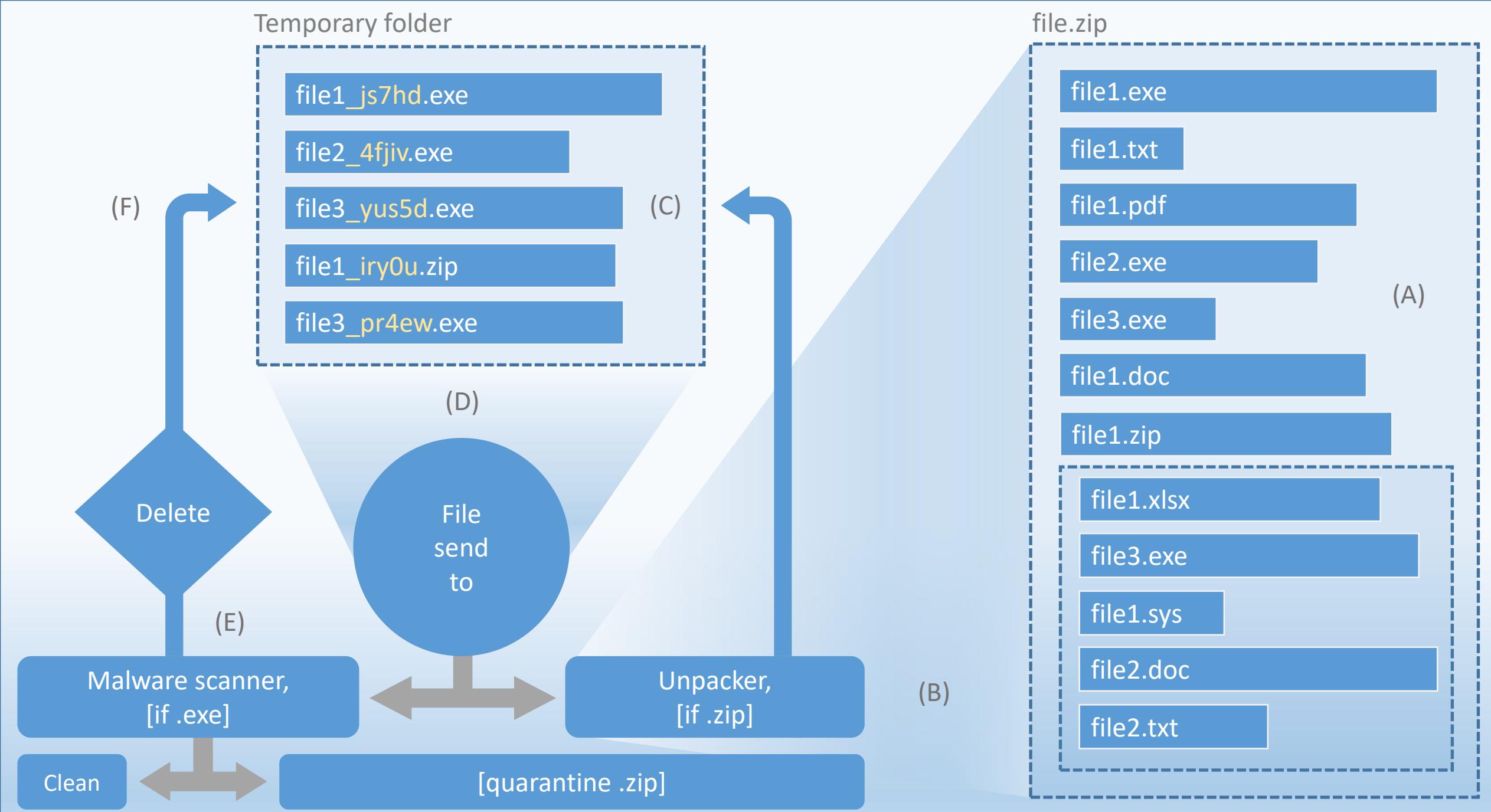












RAM

Persistent processes

Wach Dog

Wach Dog

Wach Dog

Pro-active

Wach Dog

Engine

SysTry

Temporary

Manual scan

Statistics

Desinfection

Other tools

GUI

Update

HDD/SSD

other.non-exe.files

signature.sig

Pro-active

Other tools

Desinfection

GUI

Wach Dog

SysTry

Engine

Manual scan

Statistics

Update

RAM

Persistent processes

Wach Dog

Wach Dog

Wach Dog

Pro-active

Wach Dog

Engine

SysTry

Temporary

Manual scan

Desinfection

Statistics

Other tools

GUI

Update

HDD/SSD

other.non-exe.files

signature.sig

Pro-active

Other tools

Desinfection

GUI

Wach Dog

SysTry

Engine

Manual scan

Statistics

Update

RAM

Persistent processes

Wach Dog

Wach Dog

Wach Dog

Pro-active

Wach Dog

Engine

SysTry

Temporary

Manual scan

Desinfection

Statistics

Other tools

GUI

Update

HDD/SSD

other.non-exe.files

signature.sig

Pro-active

Other tools

Desinfection

GUI

Wach Dog

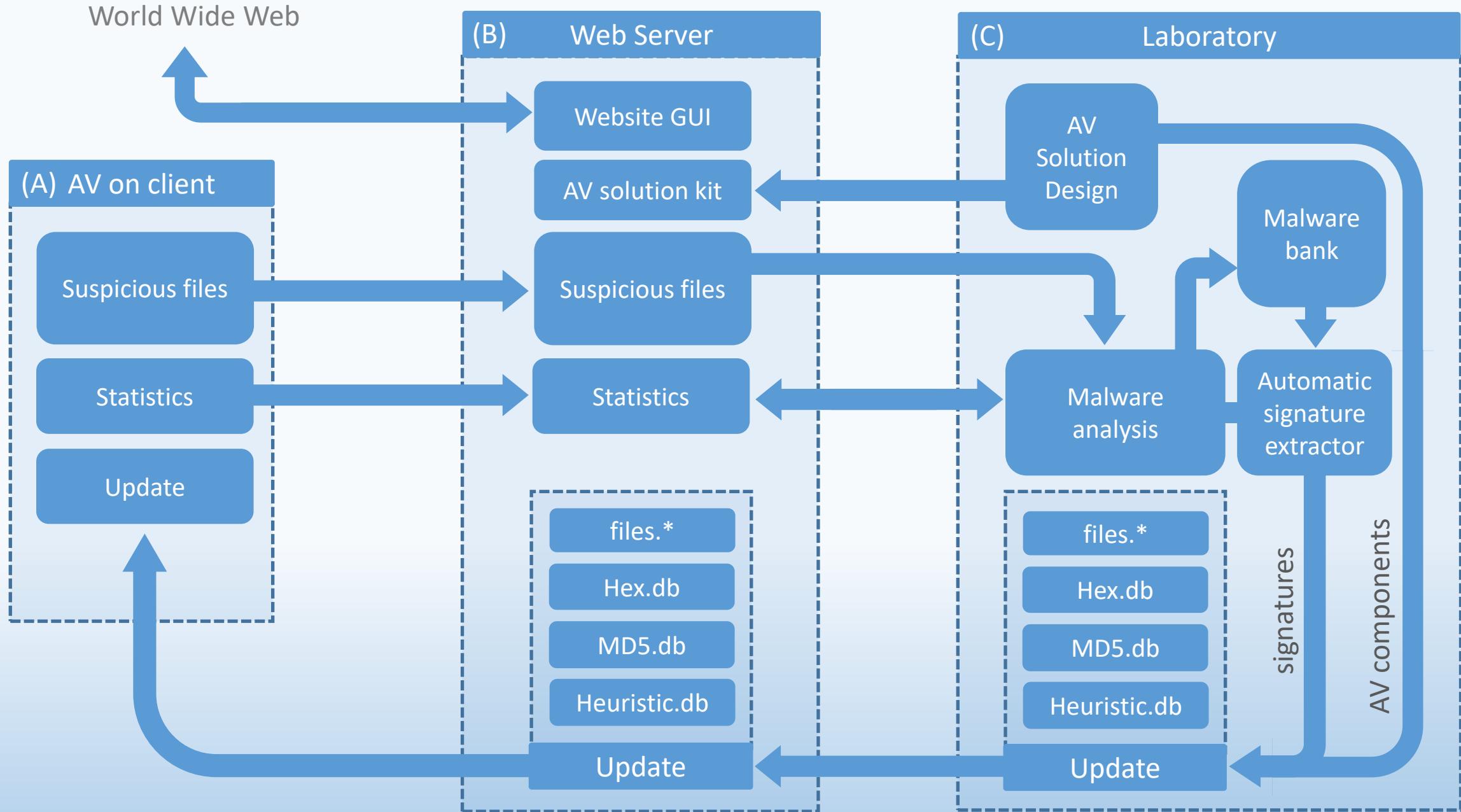
SysTry

Engine

Manual scan

Statistics

Update



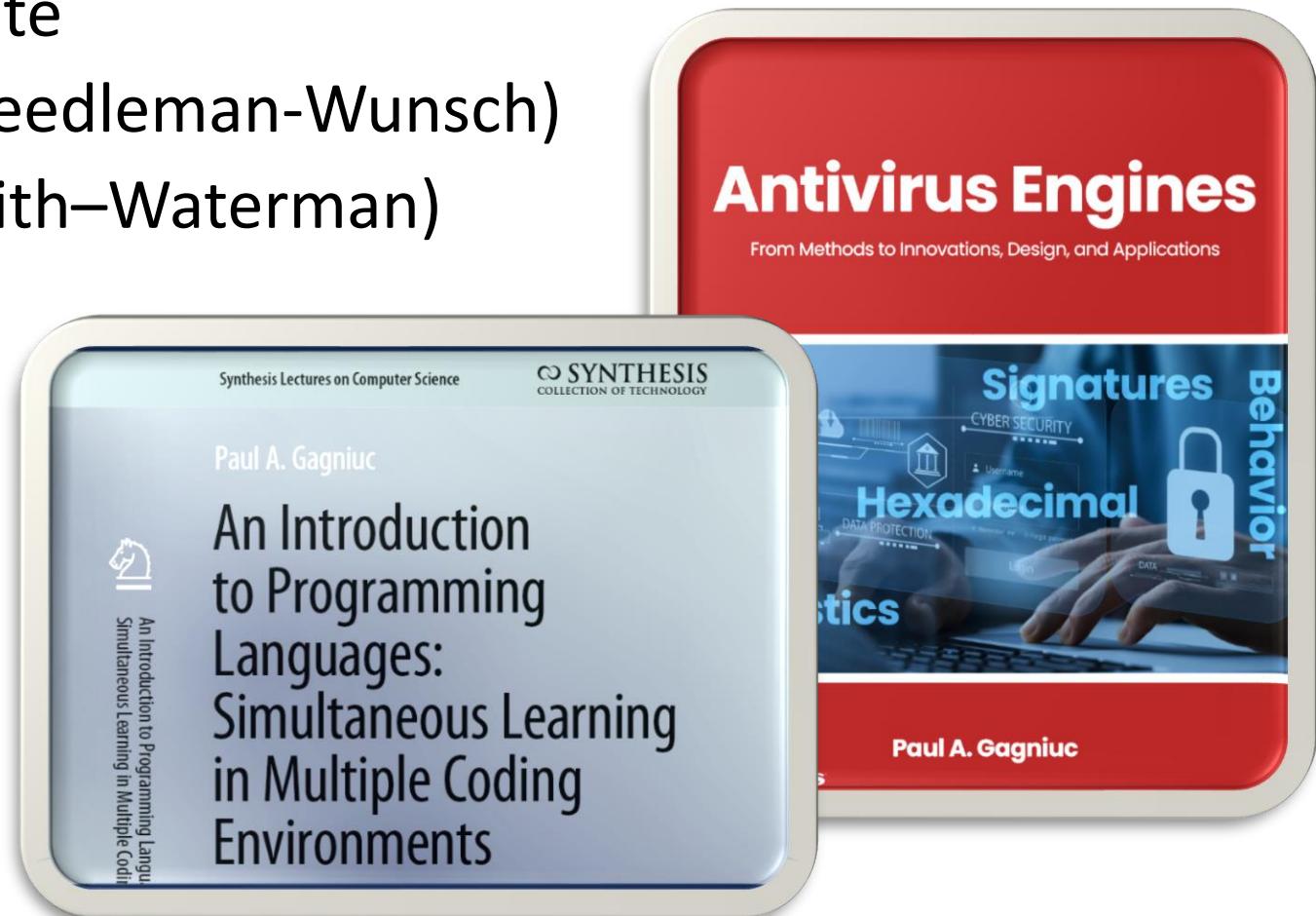
C.14.4

ALGORITMI FOLOSITI IN SECURITATEA CIBERNETICA



Algoritmi importanti in securitatea cibernetica cu aplicare directa in motoare de detectie malware:

- Algoritmi de alinieri de secente
- Aliniere Globala (algoritmul Needleman-Wunsch)
- Aliniere Locala (algoritmul Smith–Waterman)
- Binary Search
- Hash tables
- Aho-Corasick
- Boyer-Moore string-search
- Filtrul Bloom
- Algoritmul de auto-aliniere



Hash tables

Tabelul de hash. Codul furnizat implementează o structură de date de bază de tabel hash în Python folosind tehnica de înlănțuire pentru rezoluția coliziunilor. Consta dintr-o clasă HashTable care reprezintă tabelul hash și conține metode de inserare a perechilor cheie-valoare și de căutare a valorilor asociate cheilor. Clasa HashTable are un constructor care inițializează tabelul hash cu o dimensiune specificată. Acesta creează o listă goală de liste (găleți) pentru a stoca perechile cheie-valoare. O metodă privată `_hash_function(self, key)` calculează valoarea hash pentru o anumită cheie folosind funcția încorporată Python `hash()` și operatorul modulo pentru a se potrivi cu valoarea în dimensiunea tabelului hash. Metoda `insert(self, key, value)` adaugă o pereche cheie-valoare în tabelul hash. Acesta calculează indexul compartimentului folosind `_hash_function()` și adaugă perechea la compartimentul corespunzător. Metoda `search(self, key)` caută o valoare asociată cu o anumită cheie în tabelul hash. Acesta calculează indexul compartimentului folosind `_hash_function()` și iterează prin perechile din compartiment pentru a găsi cheia potrivită. Dacă este găsit, returnează valoarea asociată; în caz contrar, returnează Niciunul. În practică, o instanță a clasei HashTable este creată și perechile cheie-valoare sunt inserate folosind metoda `insert()`. Metoda `search()` este apoi utilizată pentru a prelua valori bazate pe chei. Codul demonstrează principiile de bază ale unui tabel hash, inclusiv stocarea și preluarea eficientă a perechilor cheie-valoare prin funcția hash și tehnica de înlănțuire pentru gestionarea coliziunilor hash.



```
class HashTable:  
    def __init__(self, size):  
        self.size = size  
        self.table = [[] for _ in range(size)]  
  
    def _hash_function(self, key):  
        return hash(key) % self.size  
  
    def insert(self, key, value):  
        index = self._hash_function(key)  
        self.table[index].append((key, value))  
  
    def search(self, key):  
        index = self._hash_function(key)  
        for stored_key, value in self.table[index]:  
            if stored_key == key:  
                return value  
        return None  
  
# create a hash table and insert some key-value pairs:  
hash_table = HashTable(size=10)  
hash_table.insert("d41d8cd98f00b204e9800998ecf8427e", "Trojan")  
hash_table.insert("098f6bcd4621d373cade4e832627b4f6", "Rootkit")  
hash_table.insert("1f0e3dad99908345f7439f8ffabdfffc4", "Bootkit")  
  
# search for values using keys:  
print(hash_table.search("d41d8cd98f00b204e9800998ecf8427e"))  
print(hash_table.search("098f6bcd4621d373cade4e832627b4f6"))  
print(hash_table.search("1f0e3dad99908345f7439f8ffabdfffc4"))  
print(hash_table.search("c4ca4238a0b923820dcc509a6f75849b"))
```

Output:

```
Trojan  
Rootkit  
Bootkit  
None
```

Aho-Corasick

Algoritmul Aho-Corasick. În acest exemplu, clasa AhoCorasickNode reprezintă un nod în încercarea Aho-Corasick. Clasa AhoCorasick construiește trie și se ocupă de construcția tranzițiilor de eșec. Metoda este folosită pentru a căuta modele în textul dat. Codul furnizat de mai sus implementează algoritmul Aho-Corasick, o tehnică rapidă și eficientă de potrivire a sirurilor folosită pentru a identifica mai multe modele predefinite (cuvinte cheie) într-un text dat. Algoritmul este util în special pentru sarcinile care implică căutarea mai multor cuvinte cheie în corpuși mari de text. Codul este format din două clase principale. a) Clasa AhoCorasickNode reprezintă noduri în structura trie utilizată de algoritm. Fiecare nod conține informații despre tranzițiile către alte noduri, modele de ieșire asociate nodului și un indicator de stare de eșec care ajută la traversare. b) Clasa AhoCorasick este responsabilă de implementarea algoritmului. Include următoarele metode: i) Metoda `__init__(self, patterns)` inițializează algoritmul prin construirea trie-ului folosind o listă de modele de intrare. ii) Metoda `build_trie(self, patterns)` construiește structura trie prin adăugarea iterativă a fiecărui model caracter cu caracter. iii) Metoda `build_failures(self)` construiește tranziții de eșec pentru noduri, creând legături către stările de eșec corespunzătoare. iv) Metoda `search(self, text)` căută modele în textul de intrare folosind trie construit și returnează o listă de modele potrivite. În exemplul de utilizare furnizat, este definită o listă de modele `["sig1", "sig2", "sig3", "sig4"]`, iar textul „IAsig3mAfile” este folosit pentru testare. O instanță a clasei AhoCorasick este creată cu modelele, iar metoda de căutare este invocată pentru a găsi modele în text. Modelele potrivite sunt apoi imprimate.



```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.output = []

def build_trie(patterns):
    root = TrieNode()
    for pattern in patterns:
        node = root
        for char in pattern:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.output.append(pattern)
    return root

def aho_corasick(text, root):
    results = []
    node = root
    for i, char in enumerate(text):
        while char not in node.children and node != root:
            node = node.fail
        if char in node.children:
            node = node.children[char]
        for pattern in node.output:
            results.append((i - len(pattern) + 1, pattern))
    return results

def build_fail_links(root):
    queue = []
    for node in root.children.values():
        node.fail = root
        queue.append(node)
    while queue:
        current = queue.pop(0)
        for char, child in current.children.items():
            queue.append(child)
            fail_state = current.fail
            while char not in fail_state.children and fail_state != root:
                fail_state = fail_state.fail
            child.fail = fail_state.children.get(char, root)

if __name__ == "__main__":
    patterns = ["sig1", "sig2", "sig3", "sig4"]
    text = "IAsig3mAfile"

    root = build_trie(patterns)
    build_fail_links(root)
    results = aho_corasick(text, root)

    if results:
        print("Found patterns:")
        for position, pattern in results:
            print(f"Pattern '{pattern}' found at position {position}")
    else:
        print("No patterns found.")
```

Output:

```
Found patterns:
Pattern 'sig3' found at position 2
```

Filtrul Bloom

Codul din dreapta definește o clasă Python numită BloomFilter care implementează o structură de date de bază a filtrului Bloom. Un filtru Bloom este o structură de date probabilistică utilizată pentru testarea apartenenței, în special pentru a verifica eficient dacă un element este sau nu membru al unui set. Clasa BloomFilter este inițializată cu doi parametri: dimensiune și hash_functions. size determină dimensiunea (numărul de biți) a matricei de biți de filtru Bloom, reprezentând capacitatea sa de a stoca elemente. hash_functions este o listă de funcții hash utilizate de filtru pentru hasharea elementelor. Mai multe funcții hash sunt folosite pentru a distribui elementele în matricea de biți. Metoda constructorului (`__init__`) inițializează filtrul Bloom setându-i dimensiunea, creând o matrice de biți (inițializată cu zerouri) și stochând funcțiile hash furnizate. Metoda de adăugare vă permite să adăugați elemente la filtrul Bloom. Pentru fiecare funcție hash din lista hash_functions, calculează un index în matricea de biți prin aplicarea funcției hash la element și luând modulo rezultat cu dimensiunea filtrului. Apoi setează bitul corespunzător din matricea de biți la 1, indicând faptul că acest element este prezent (sau ar putea fi prezent) în filtru. Metoda `__contains__` supraîncarcă operatorul `in`, permitând cuiva să verifice dacă un element este prezent în filtru. Pentru fiecare funcție hash din lista hash_functions, calculează un index în matricea de biți în același mod ca metoda add. Dacă oricare dintre biții corespunzători din matricea de biți este 0, returnează imediat False, indicând că elementul cu siguranță nu este prezent în filtru. Dacă toți biții corespunzători sunt 1, returnează True, indicând faptul că elementul ar putea fi prezent în filtru. Cu toate acestea, sunt posibile rezultate false pozitive. Exemplul de utilizare demonstrează cum să creați un BloomFilter cu o dimensiune de 100 și două funcții hash (funcția hash încorporată).



```
class BloomFilter:  
    def __init__(self, size, hash_functions):  
        self.size = size  
        self.bit_array = [0] * size  
        self.hash_functions = hash_functions  
  
    def add(self, item):  
        for fn in self.hash_functions:  
            index = fn(item) % self.size  
            self.bit_array[index] = 1  
  
    def __contains__(self, item):  
        for fn in self.hash_functions:  
            index = fn(item) % self.size  
            if self.bit_array[index] == 0:  
                return False  
        return True  
  
# create a Bloom  
# filter with a size  
# of 100 and two  
# hash functions:  
bloom_filter = BloomFilter(100, [hash, hash])  
  
# add items to the filter:  
bloom_filter.add("sig1")  
bloom_filter.add("sig2")  
  
# are the items in  
# the filter already?:  
print("Is 'sig1' in the filter?", "sig1" in bloom_filter)  
print("Is 'sig3' in the filter?", "sig3" in bloom_filter)
```

Output:

```
Is 'sig1' in the filter? True  
Is 'sig3' in the filter? False
```



Antivirus Engines

From Methods to Innovations, Design, and Applications

Paul A. Gagniuc. *Antivirus Engines: From Methods to Innovations, Design, and Applications*. Cambridge, MA: Elsevier Syngress, 2024. pp. 1-656.



BIBLIOGRAFIE / RESURSE

- Paul A. Gagniuc. *Antivirus Engines: From Methods to Innovations, Design, and Applications*. Cambridge, MA: Elsevier Syngress, 2024. pp. 1-656.
- Paul A. Gagniuc. *An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments*. Synthesis Lectures on Computer Science. Springer International Publishing, 2023, pp. 1-280.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in MATLAB*, Springer, 2024, pp. 1-255.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in Python*, Springer, 2024, pp. 1-245.
- Paul A. Gagniuc. *Coding Examples from Simple to Complex - Applications in Javascript*, Springer, 2024, pp. 1-240.
- Paul A. Gagniuc. *Markov chains: from theory to implementation and experimentation*. Hoboken, NJ, John Wiley & Sons, USA, 2017, ISBN: 978-1-119-38755-8.

<https://github.com/gagniuc>

