# Memory Tracing
# Forensic Reverse Engineering

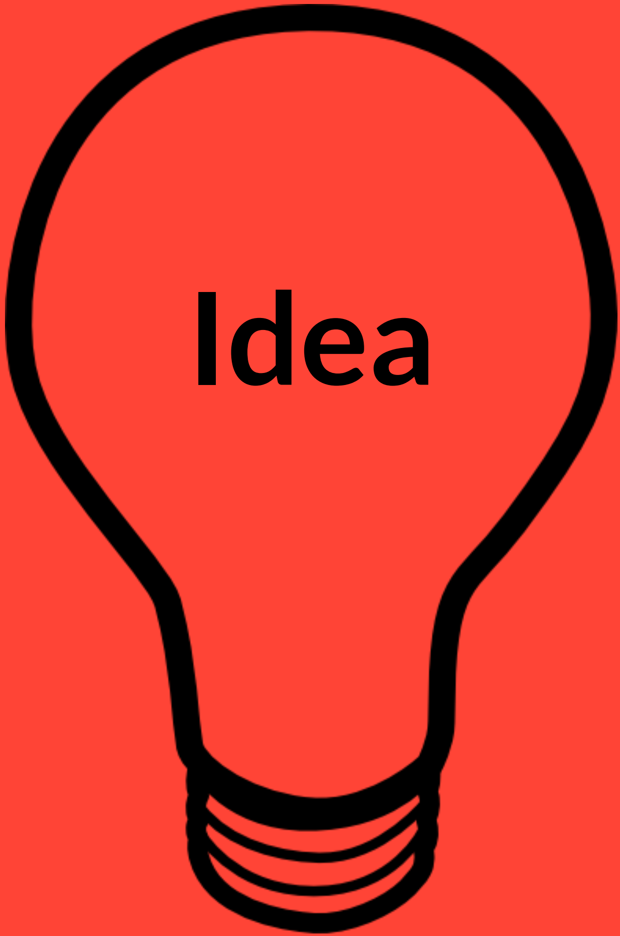
Recon 2014 Montreal

Endre Bangerter & Dominic Fischer

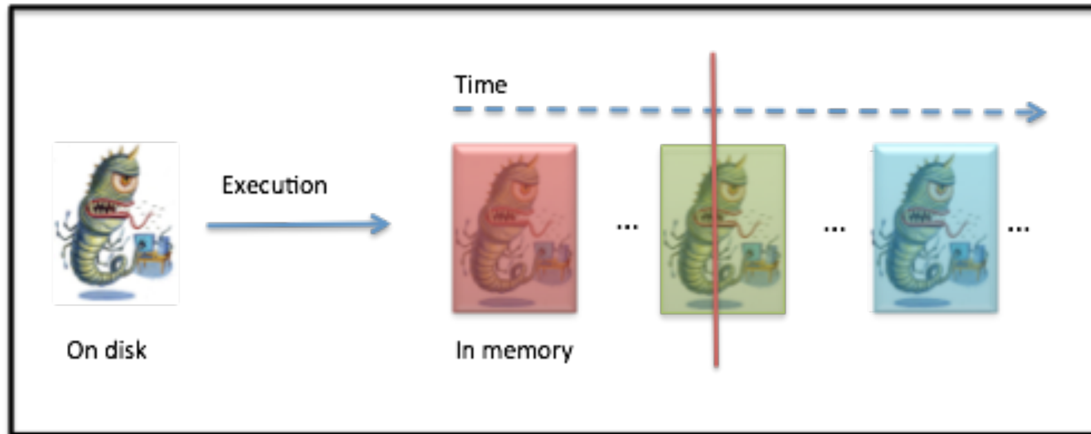
Security Engineering Lab | **http://sel.bfh.ch** | Bern University of Applied Sciences

# Thanks to

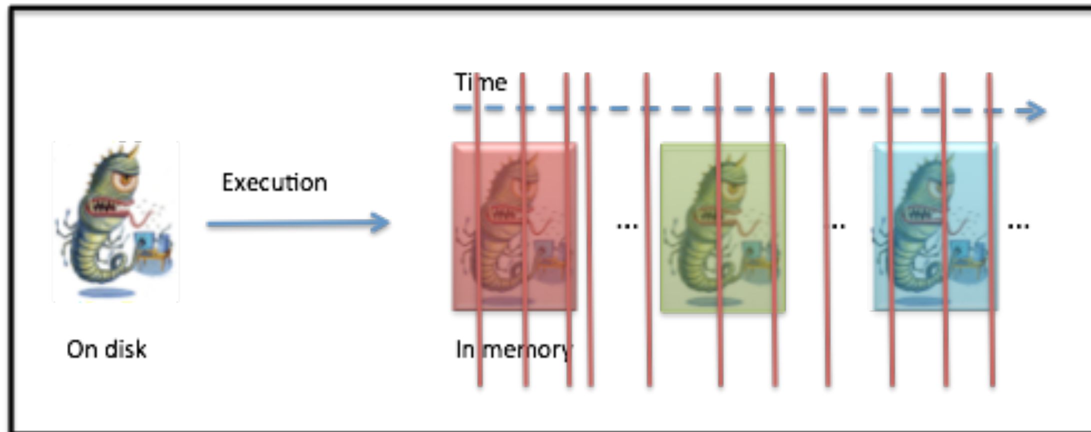
- Jonas Wagner
- Damien Schaeffer
- David Gulasch


Idea

# Memory tracing




"Traditional" memory forensics




**Memory tracing**

- **Memory trace** = series of memory snapshots

# Memory tracing, why potentially good?!?

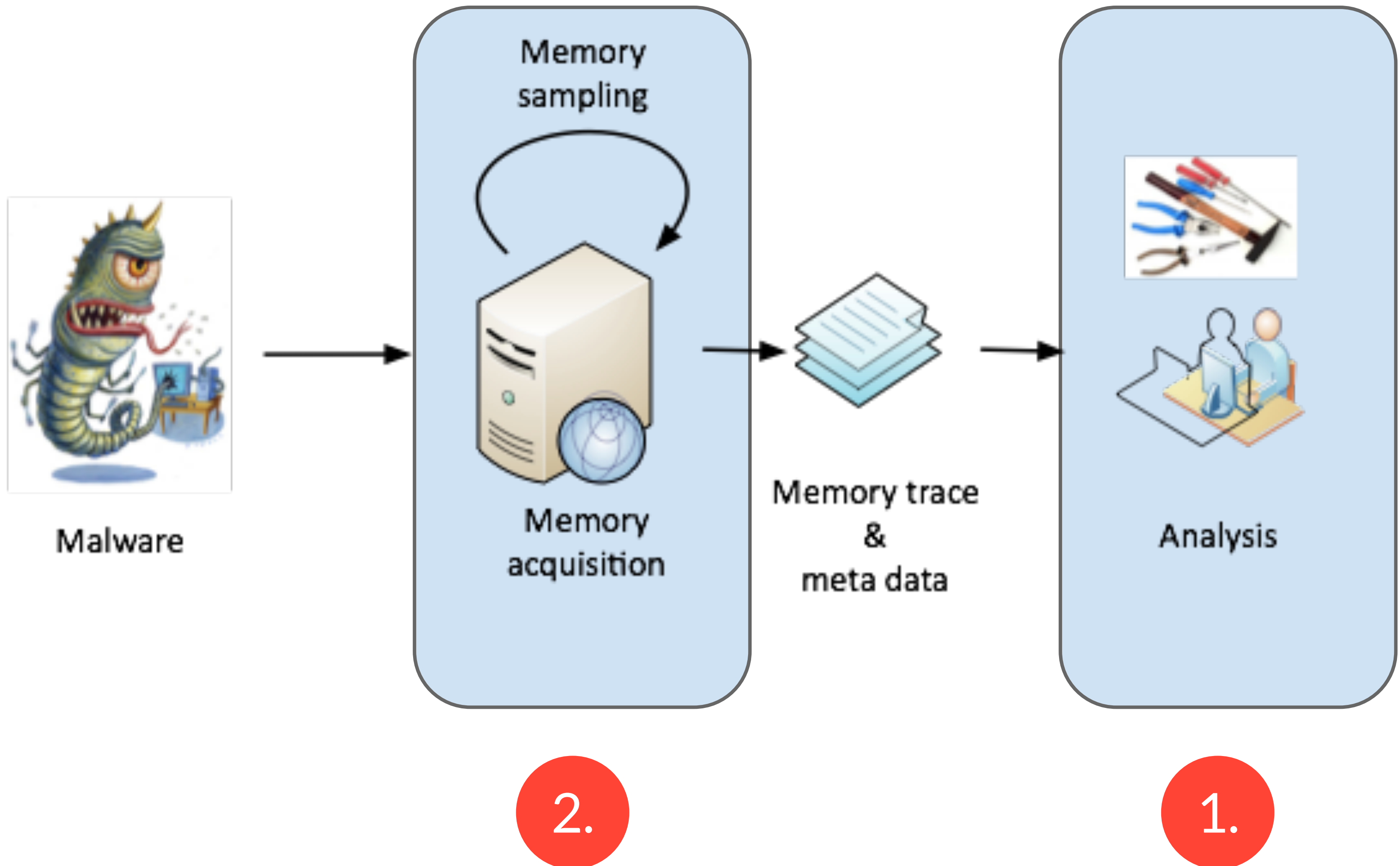
- Intuition:

  - Comprehensive capture of system behavior
  - Captures **transient** memory contents (i.e., short lived data & code)
    - Obfuscated data & code / self modifying code
    - Cypto keys & buffers
    - Short lived data:  networks buffers, URLs, config data, passwords…

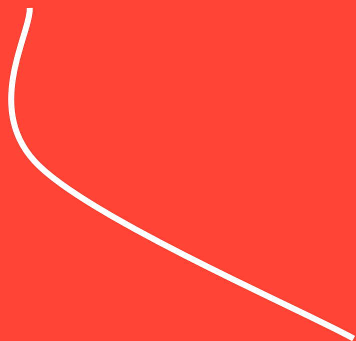
- We'll show:

  - Can be used for analysing malware
  - **Automate *some aspects*** of malware analysis
  - **Guide analysts quickly to interesting memory regions,** for further manual analysis

# The system perspective



Memory timelines

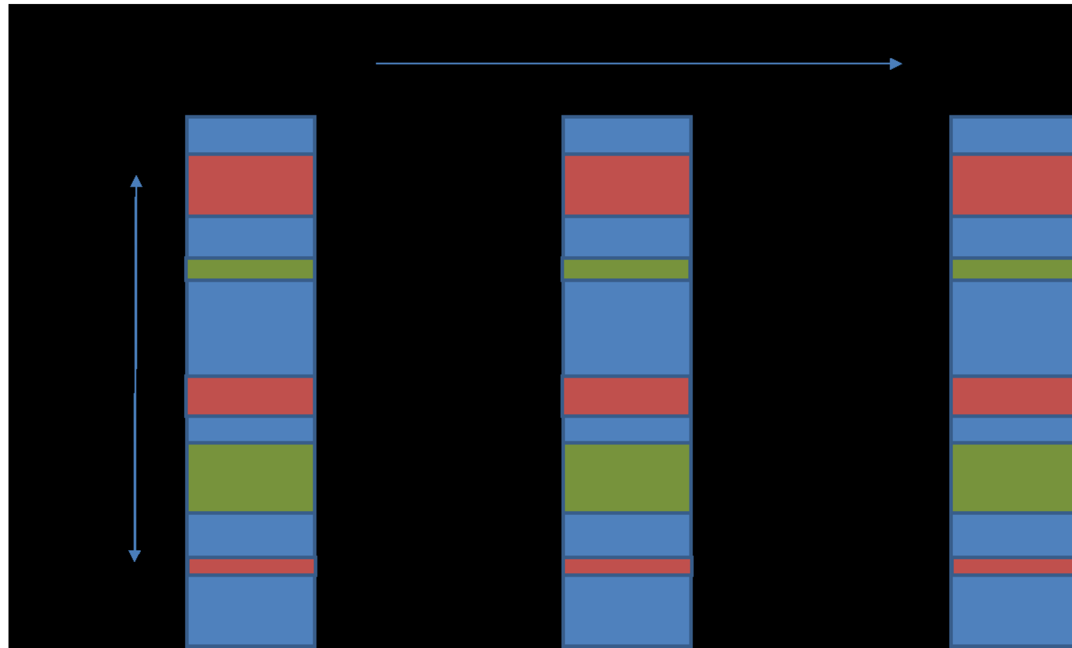


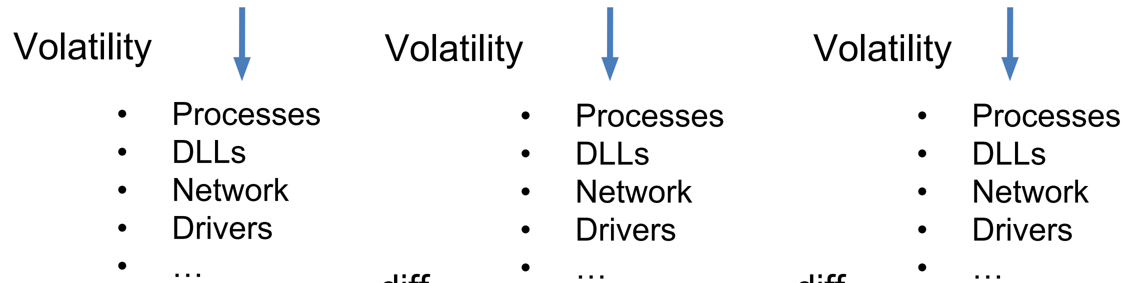
# Analysis

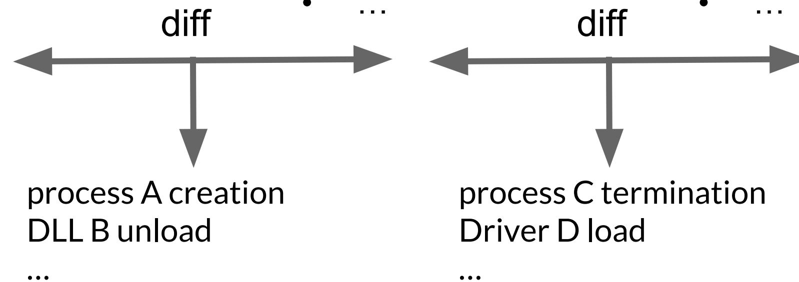
# Memory timelines

- One way to start an analysis is using sandbox report to get big picture of malware behavior
    - **File modifications**
    - **Processes started**
    - **Network activity**
    - **Registry**
    - **....**
- Let‘s see if we can recover information similar to existing sandboxes from memory traces?!?

# Memory timelines - How?





# Memory timelines - What?

- Generate list with system events
    - e.g, 4000 events for 30sec trace

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 12/20/13 | 14:47:59 | snap_020.bir | handles | + | Handles | zeroaccess_d4a | Key |
| 12/20/13 | 14:47:59 | snap_020.bir | printkey | + | Registry | Run | G |
| 12/20/13 | 14:47:59 | snap_020.bir | filescan | + | File | InstallFlashPlayer.exe | '-W----' |
| 12/20/13 | 14:47:59 | snap_020.bir | filescan | + | File | msimg32.dll | '-W----' |
| 12/20/13 | 14:47:59 | snap_020.bir | filescan | + | File | Endpoint | '-------' |
| 12/20/13 | 14:47:59 | snap_020.bir | filescan | + | File | @ | '-W----' |
| 12/20/13 | 14:47:59 | snap_020.bir | filescan | + | File | GoogleUpdate.exe | '-W----' |
| 12/20/13 | 14:47:59 | snap_020.bir | thrdscan | + | Thread | zeroaccess_d4a | PID 2064 |
| 12/20/13 | 14:47:59 | snap_019.bir | handles | - | Handles | explorer.exe | Thread |
| 12/20/13 | 14:47:59 | snap_019.bir | handles | - | Handles | zeroaccess_d4a | File |
| 12/20/13 | 14:47:59 | snap_019.bir | handles | - | Handles | zeroaccess_d4a | File |

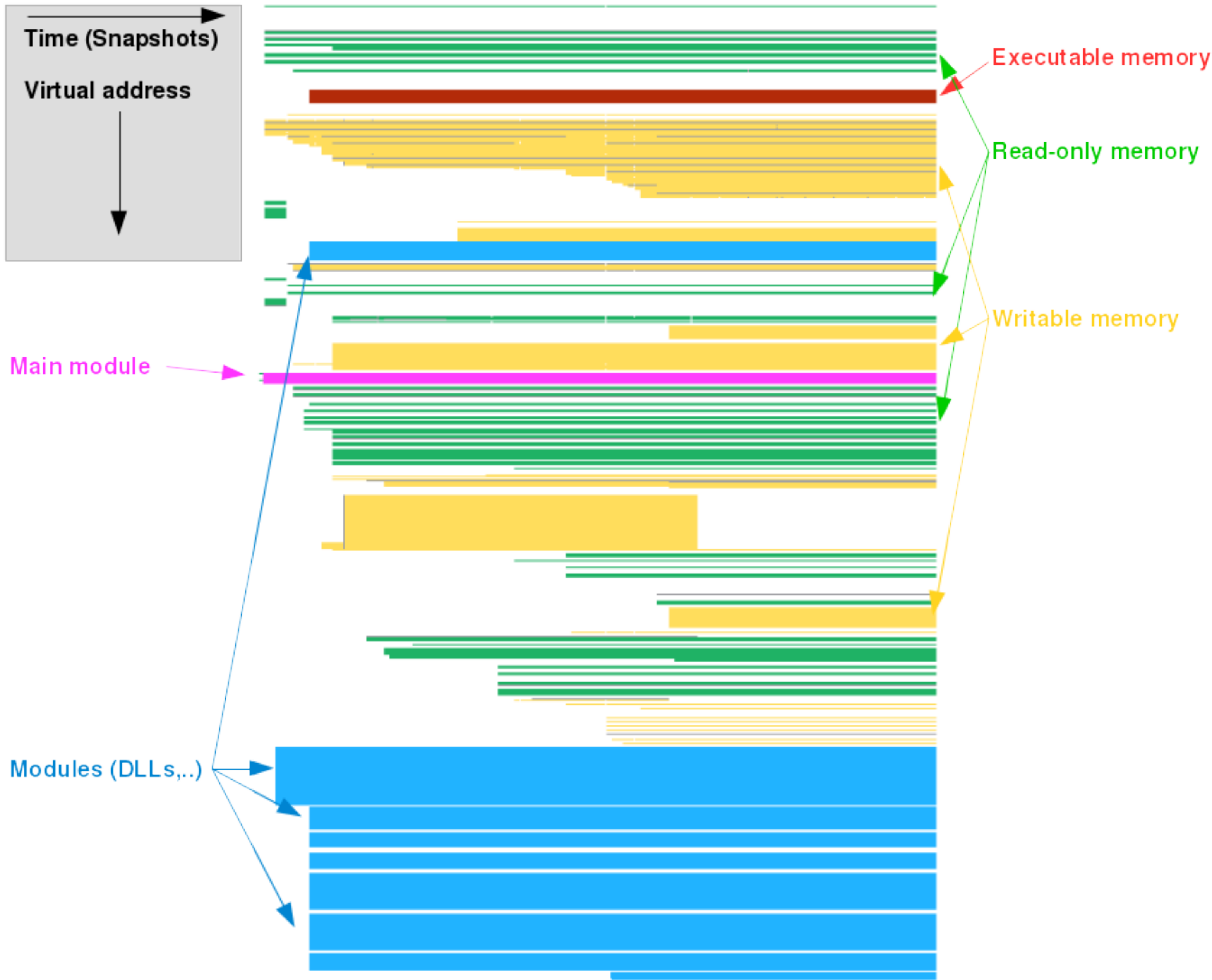
- Similar to sandboxes, but once we identify an interesting event, can look at respective snapshot to dig into details

Memory timelines

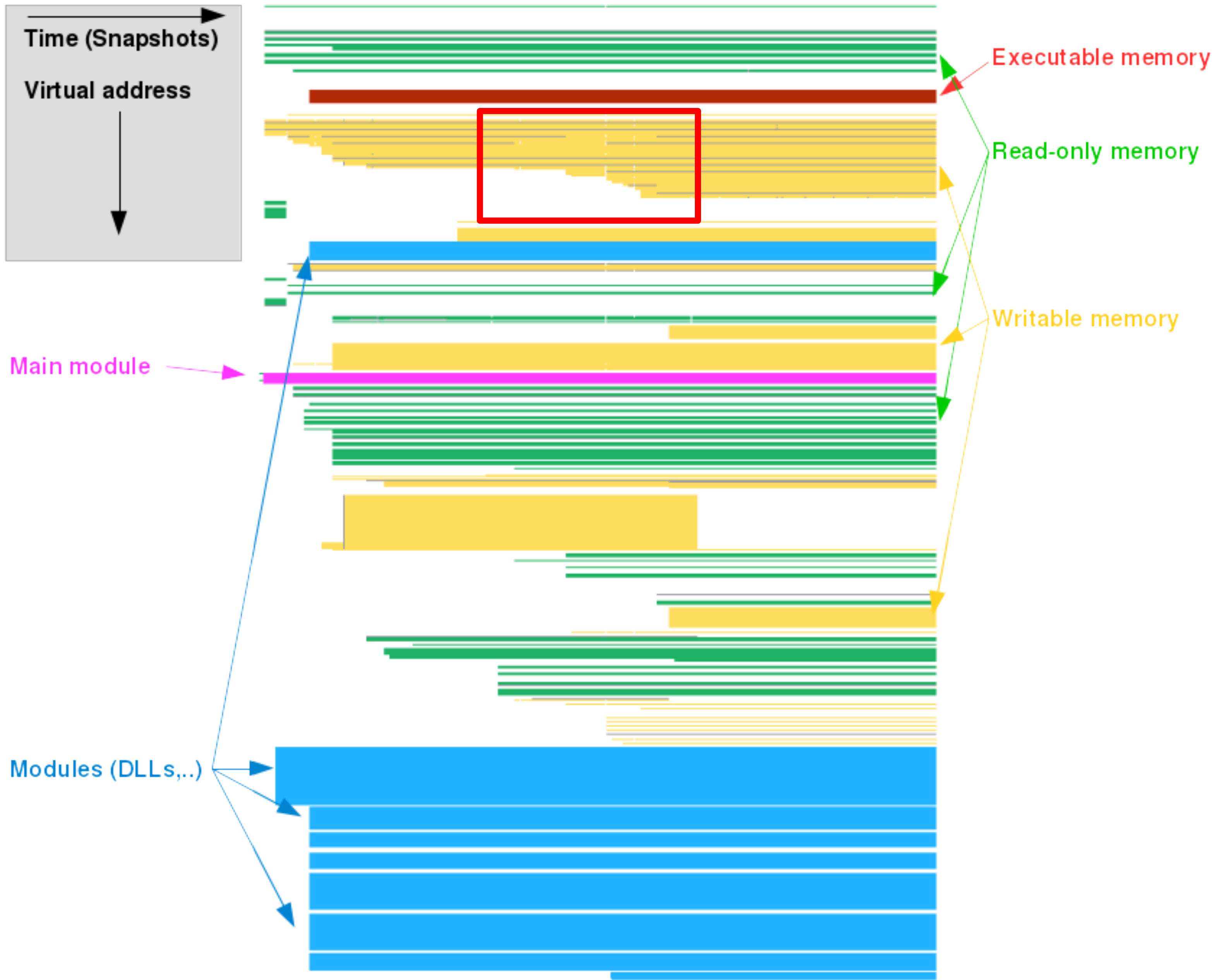
Visualization

**Analysis**

# Virtual address space



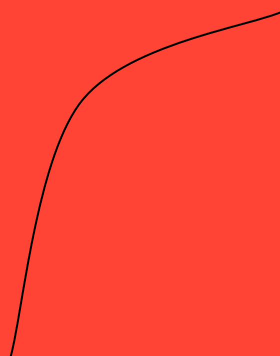
# Virtual address space



# Virtual address space - Zoomed

Memory timelines

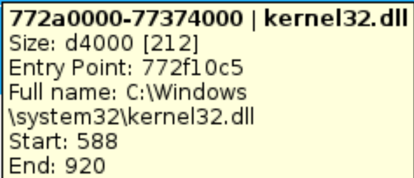
Visualization



**Whitelisting**

# Analysis

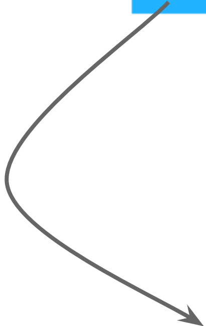
# Whitelisting

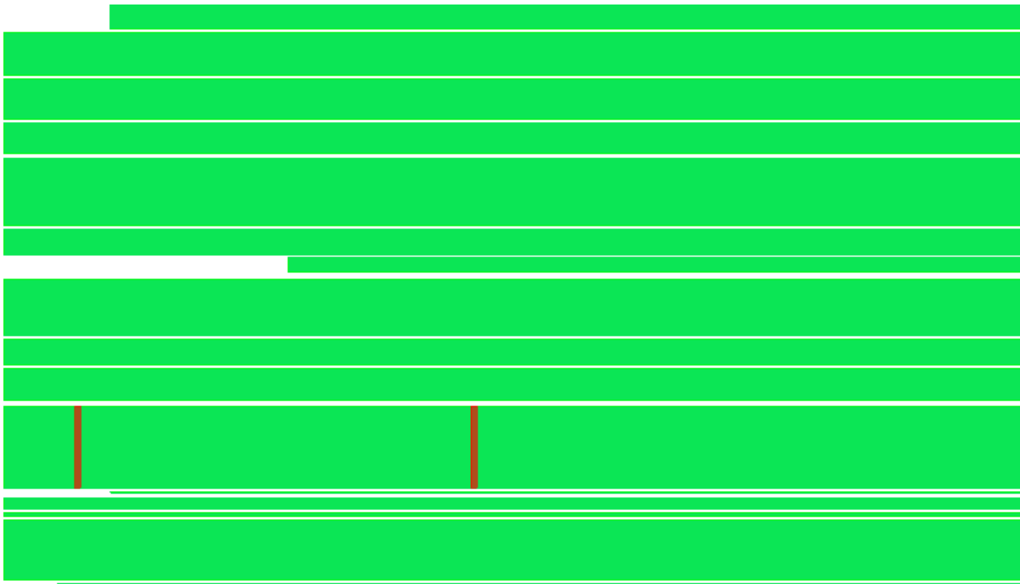


# Whitelisting

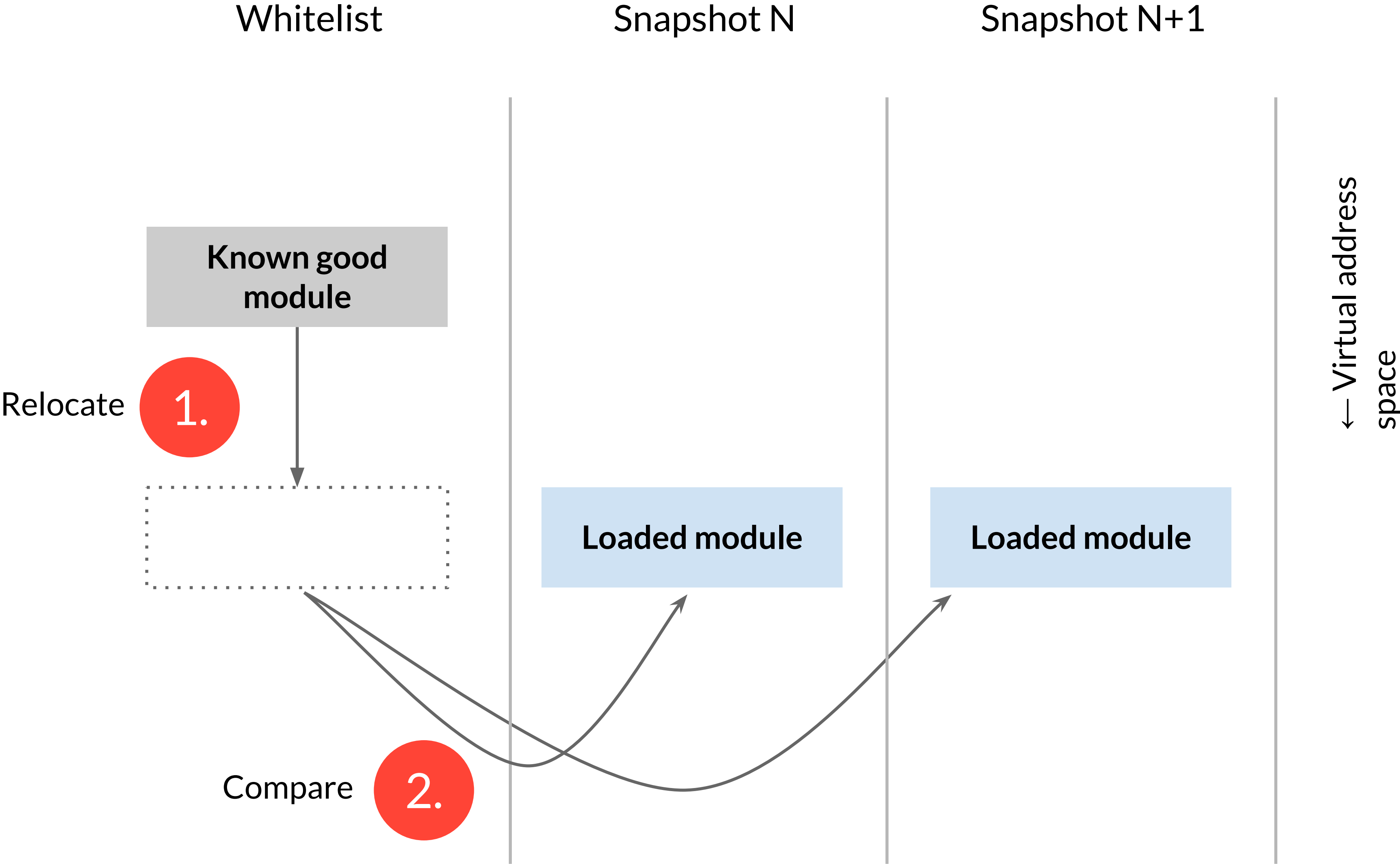
**772a0000-77374000 | kernel32.dll**
Size: d4000 [212]
Entry Point: 772f10c5
Full name: C:\Windows\system32\kernel32.dll
Start: 588
End: 920

**Whitelisting**





# Whitelisting

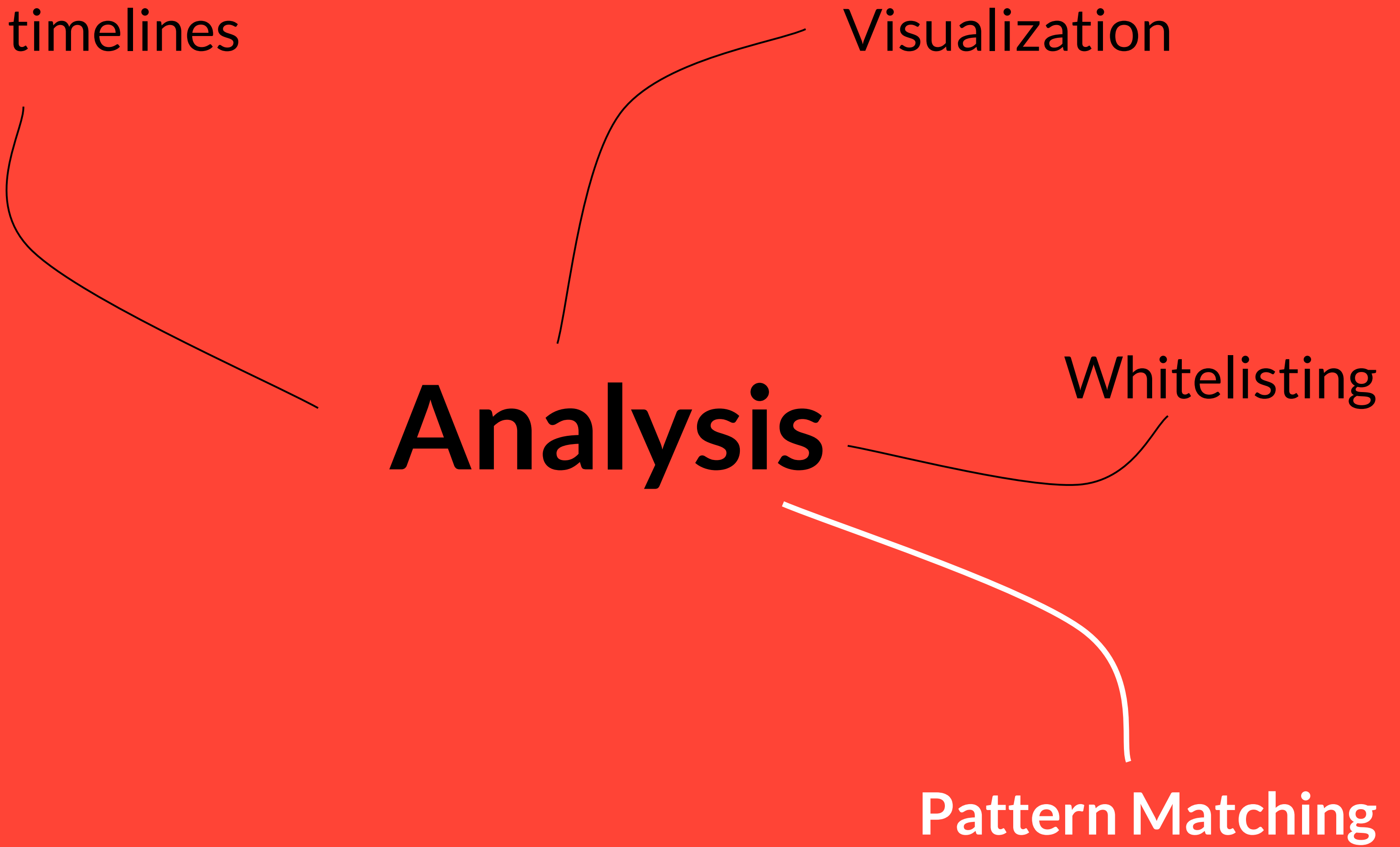


# Difficulties

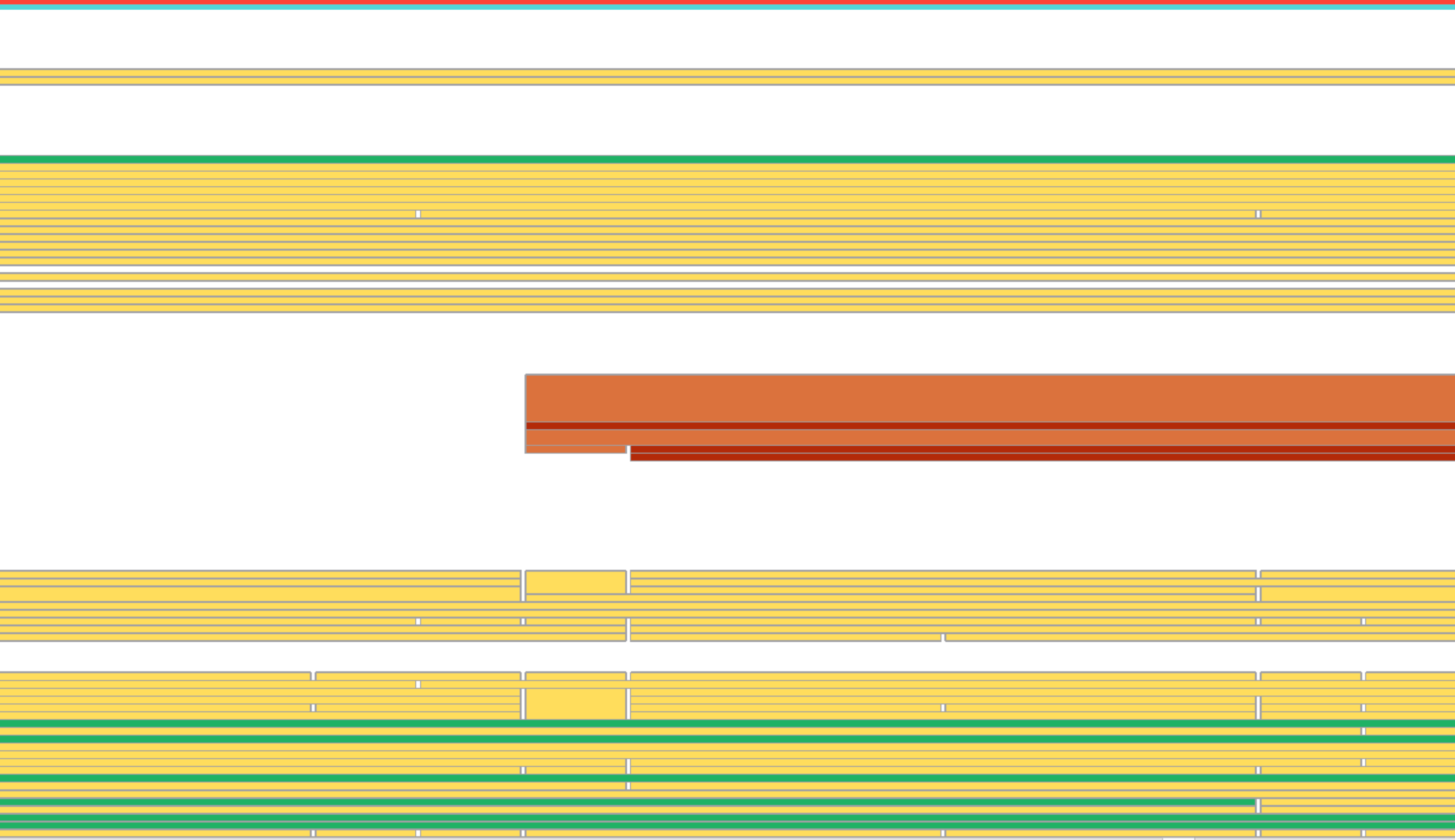
- Import Address Table (IAT)
  - contains pointers to other relocated modules

- Solution
  - Check each module without IAT
  - Check IAT of all modules at the end
    - IAT-Entry either to whitelisted module or 0x00000000

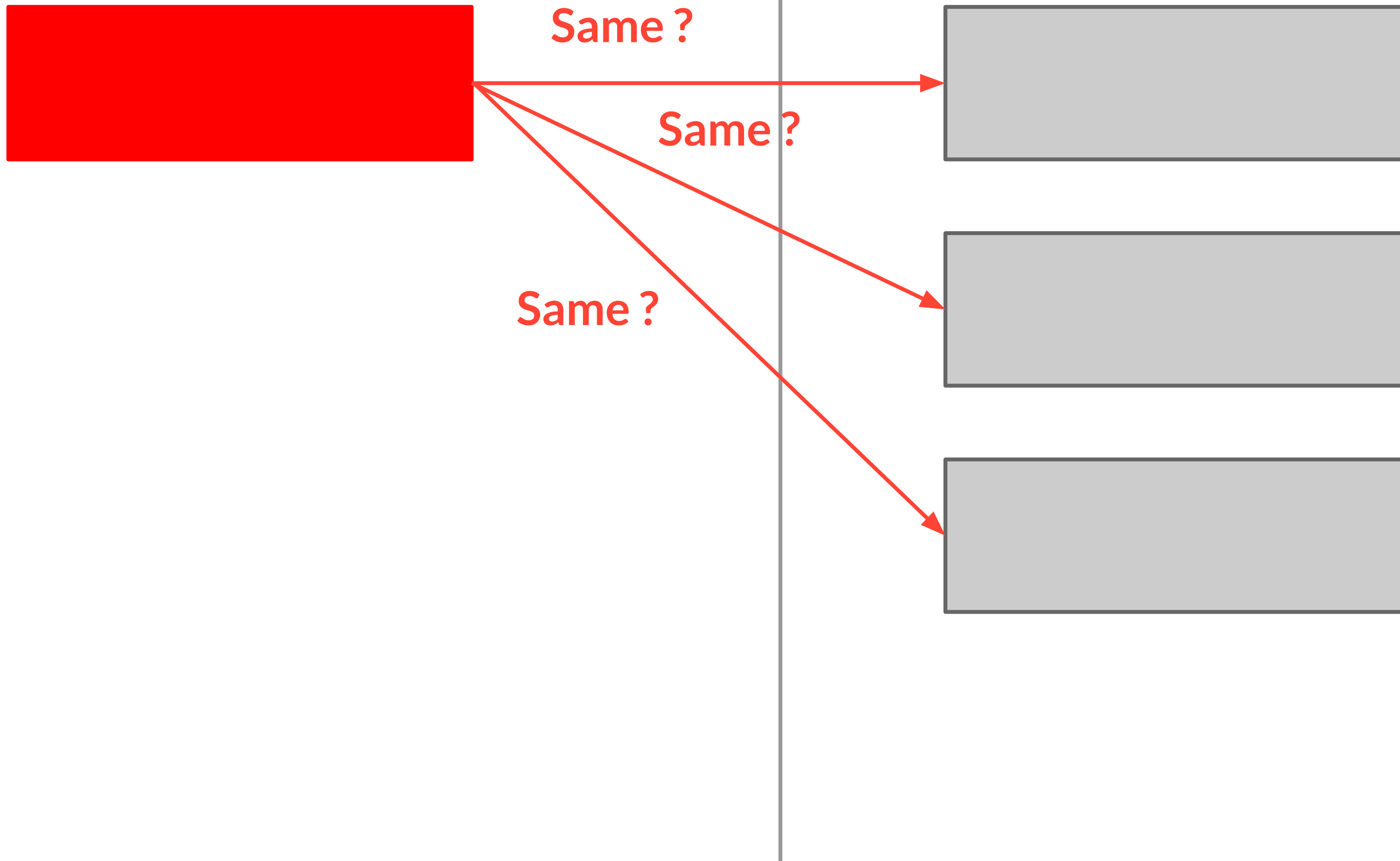
- “.orpc” sections contain self modifying code
  - Temporal solution: configurable -> ignore / check


Memory timelines
Visualization
Analysis
Whitelisting
Pattern Matching

# Pattern matching



# Pattern matching



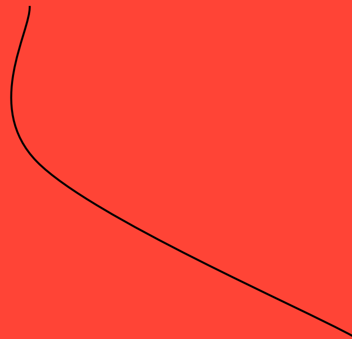
# Algorithm

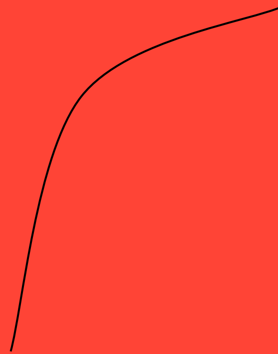
1. Reduce page data to a locality sensitive hash (LSH)
   a. Similar data results in a similar hash
   b. Easily comparable with hamming distance
   c. Avoid comparing each and every byte

2. Perform a range search
   a. Build a search tree
   b. Find matching neighbors by comparing LSHs
   c. Fast and efficient

Memory timelines

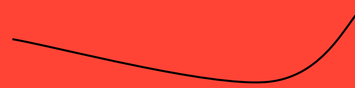
Visualization

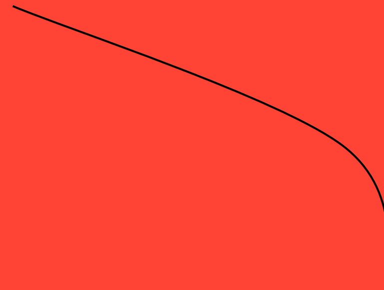




Whitelisting

# Analysis





Pattern Matching

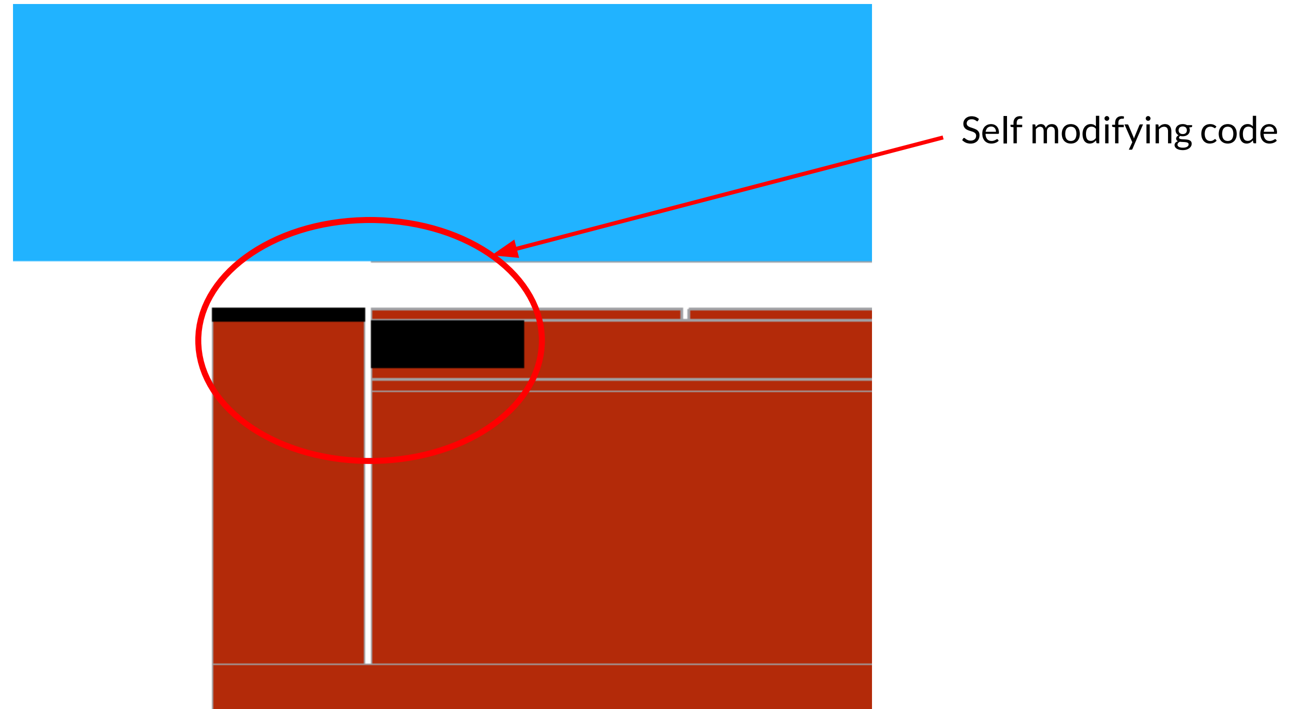
**Self modifying code (SMC)**

# Detecting self modifying code (SMC)

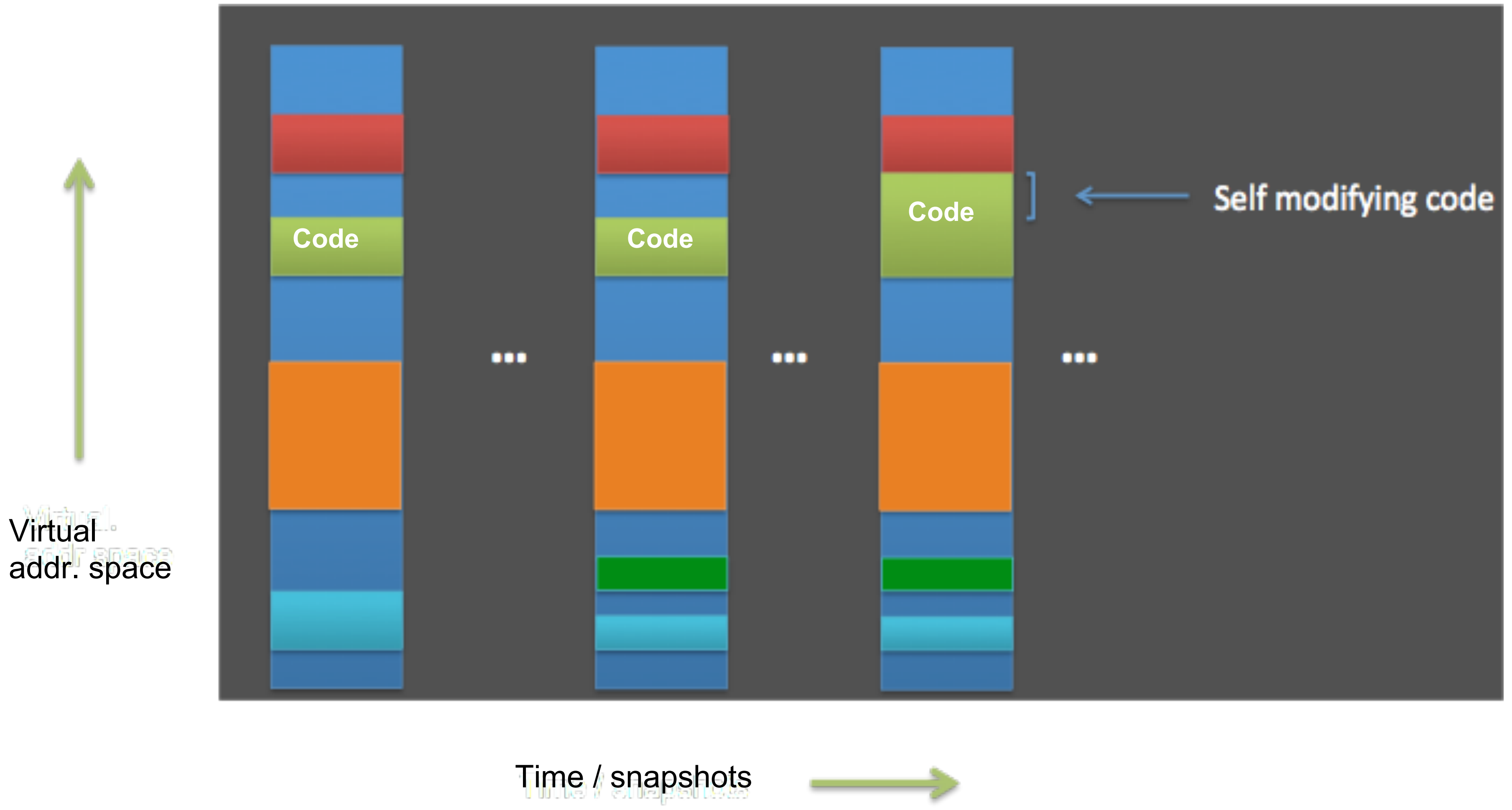
- Self modifying code (SMC) used a lot by malware
  - E.g., packing / unpacking
  - **Often contains interesting code**
  - Sometimes unpacked, then re-packed (**transient code**)

- **Goal:** Find SMC in memory trace to guide analyst to SMC quickly

# Detecting self modifying code (SMC)



# Finding SMC - How?

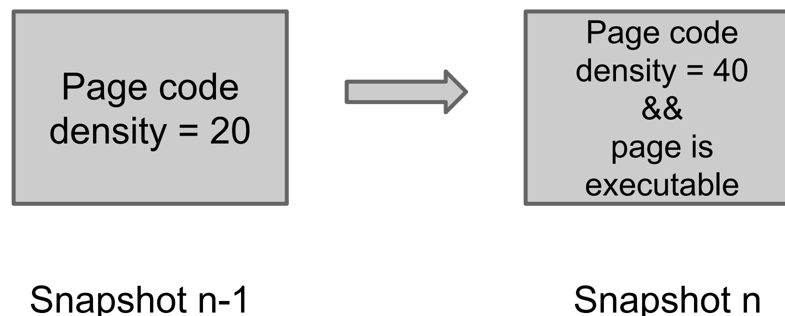


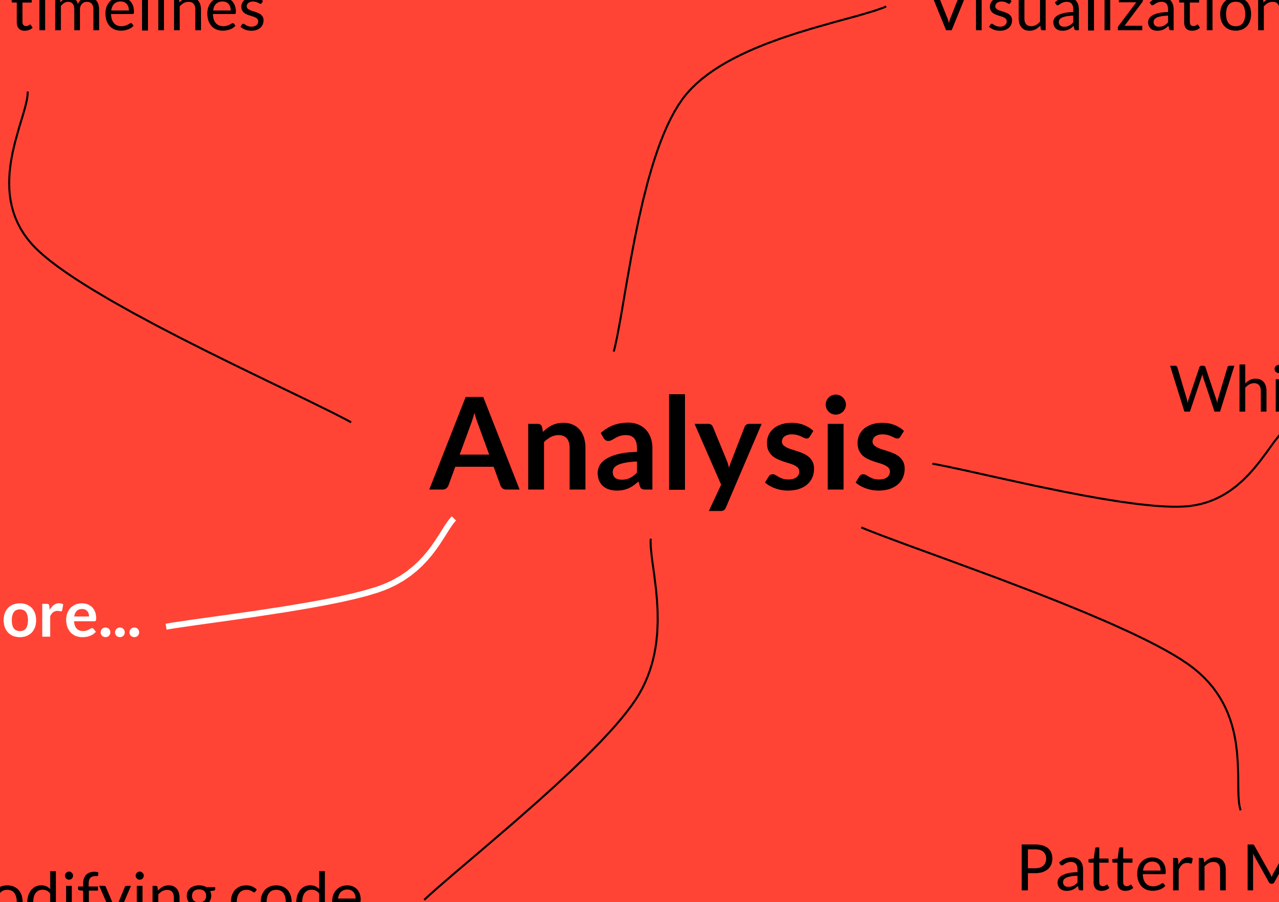
# Finding SMC - How?

- How to find code regions?

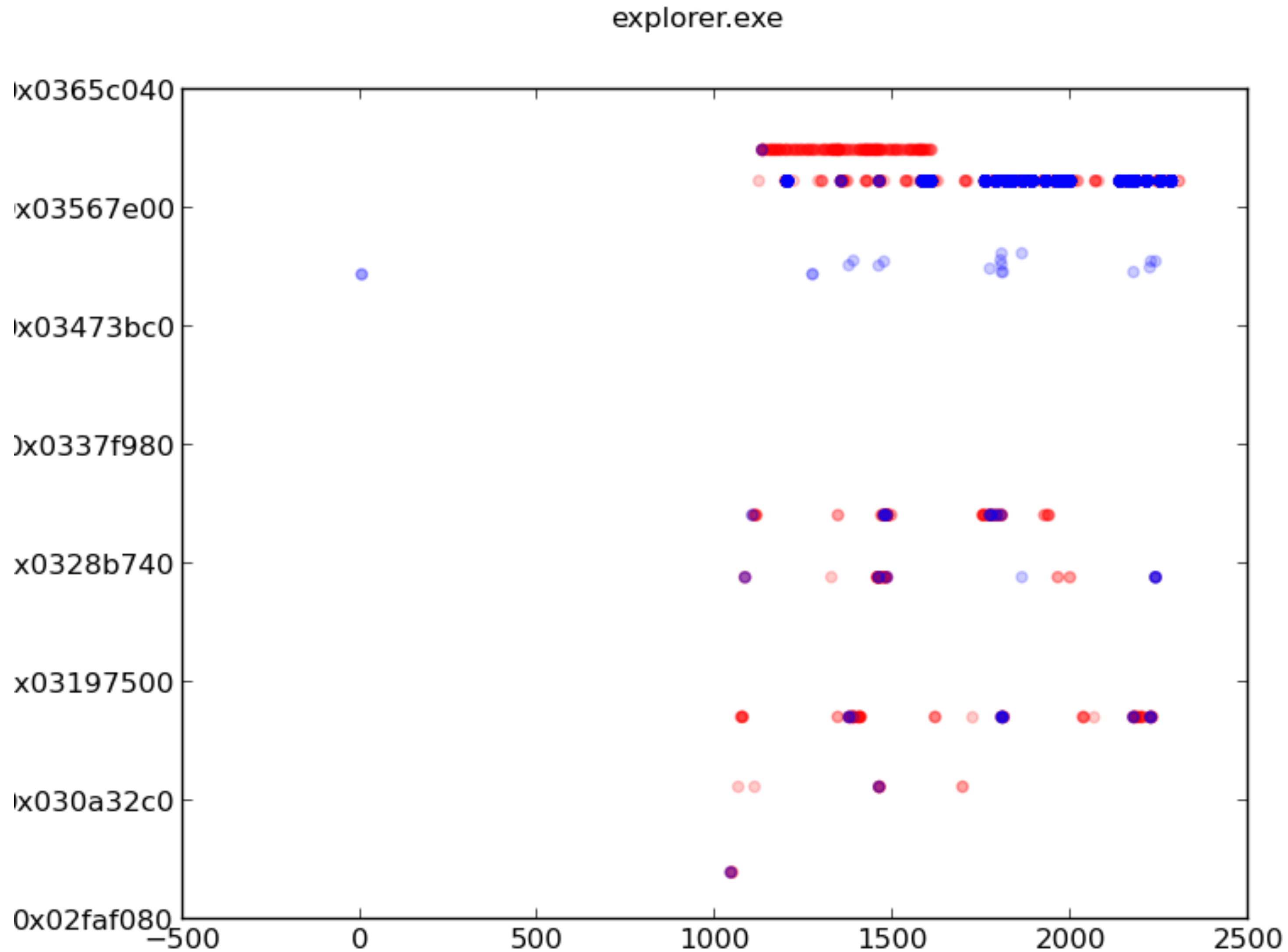
- **Very simple heuristic** to compute code density turns out to be sufficient
  - Count characteristic instructions (`CALL DWORD, PUSH DWORD, POP DWORD,...`) per page
  - Identify function prologue (`PUSH EBP; MOV EBP, ESP`) and function epilogues (`POP EBP; RET`)

- SMC is detected if code density of a page increases from one snapshot to next and if page is executable



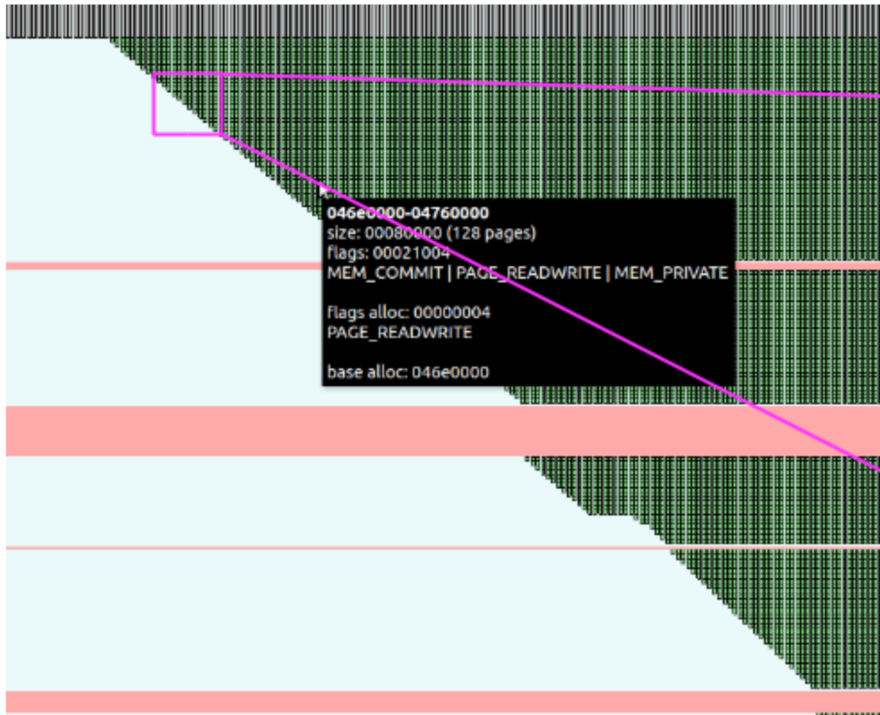

Memory timelines
Visualization
Whitelisting
Analysis
More...
Self modifying code (SMC)
Pattern Matching

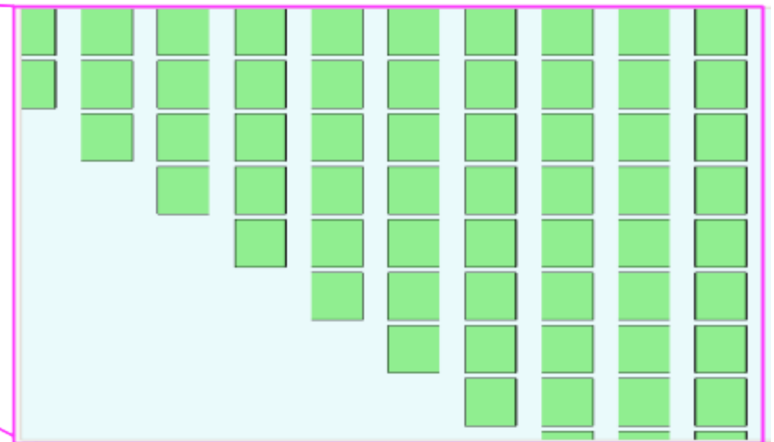
# Crypto detection – Zeus




Ongoing work with @cryptopath / Pascal Junod

# Heap spray detection

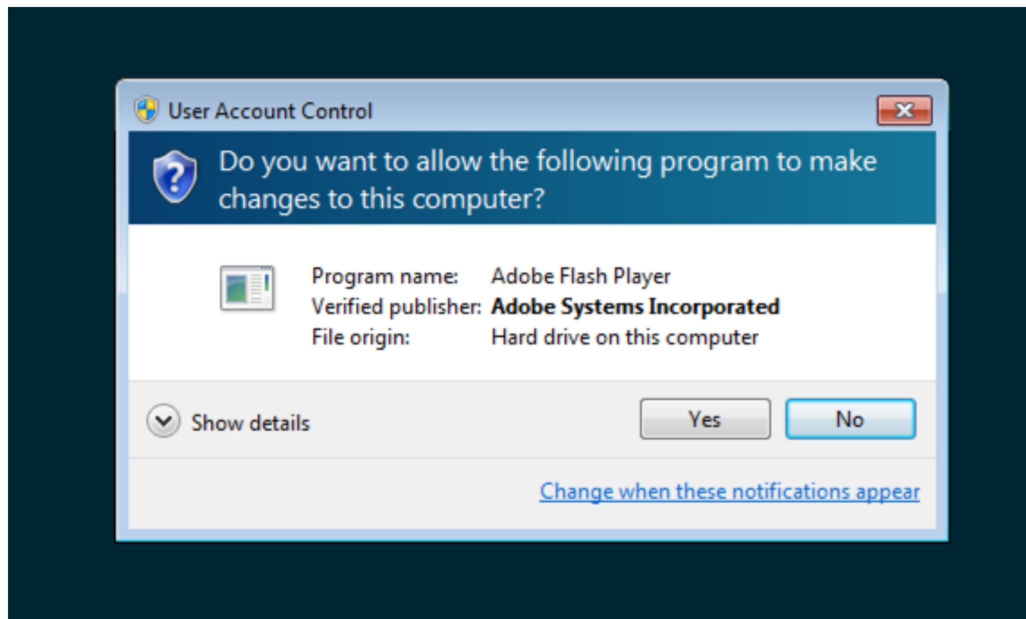


# Demo

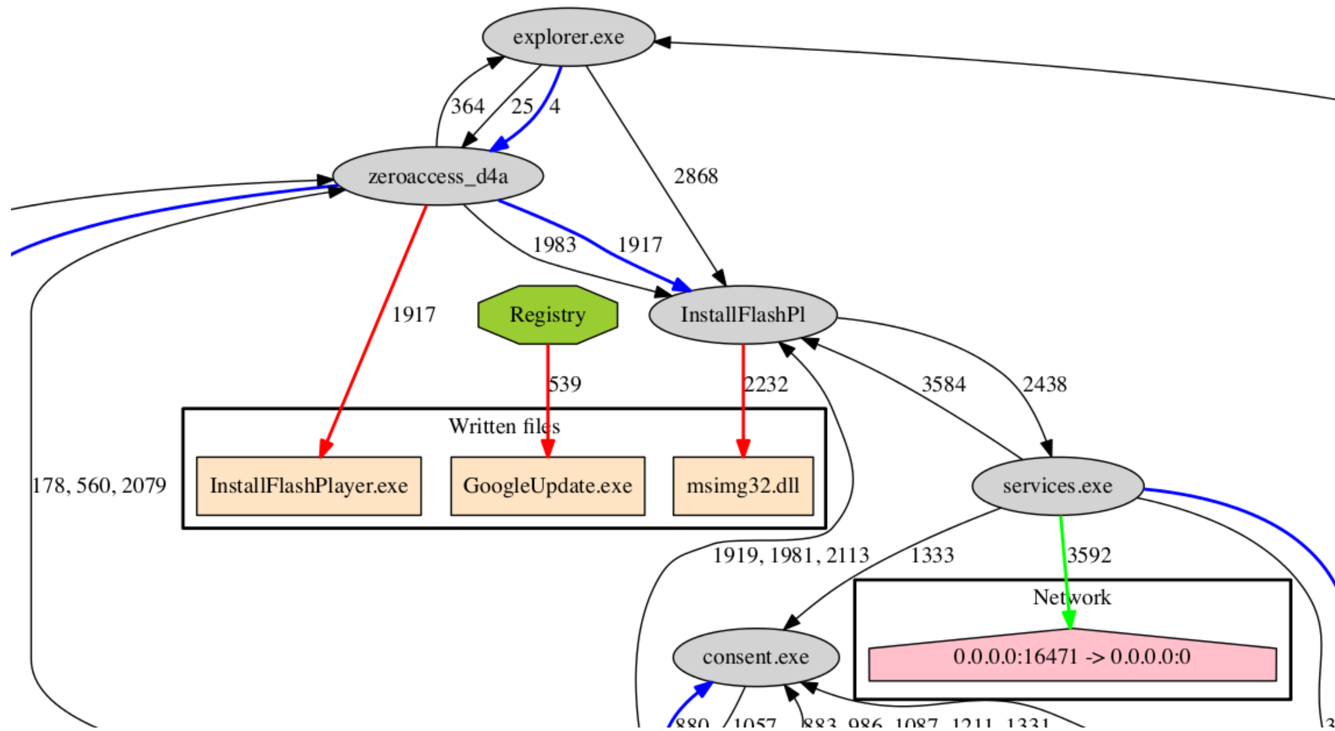
# Zeroaccess

- P2P bot
- What user sees upon infection:




- Using the analysis features shown so far, let's try to understand what's going on

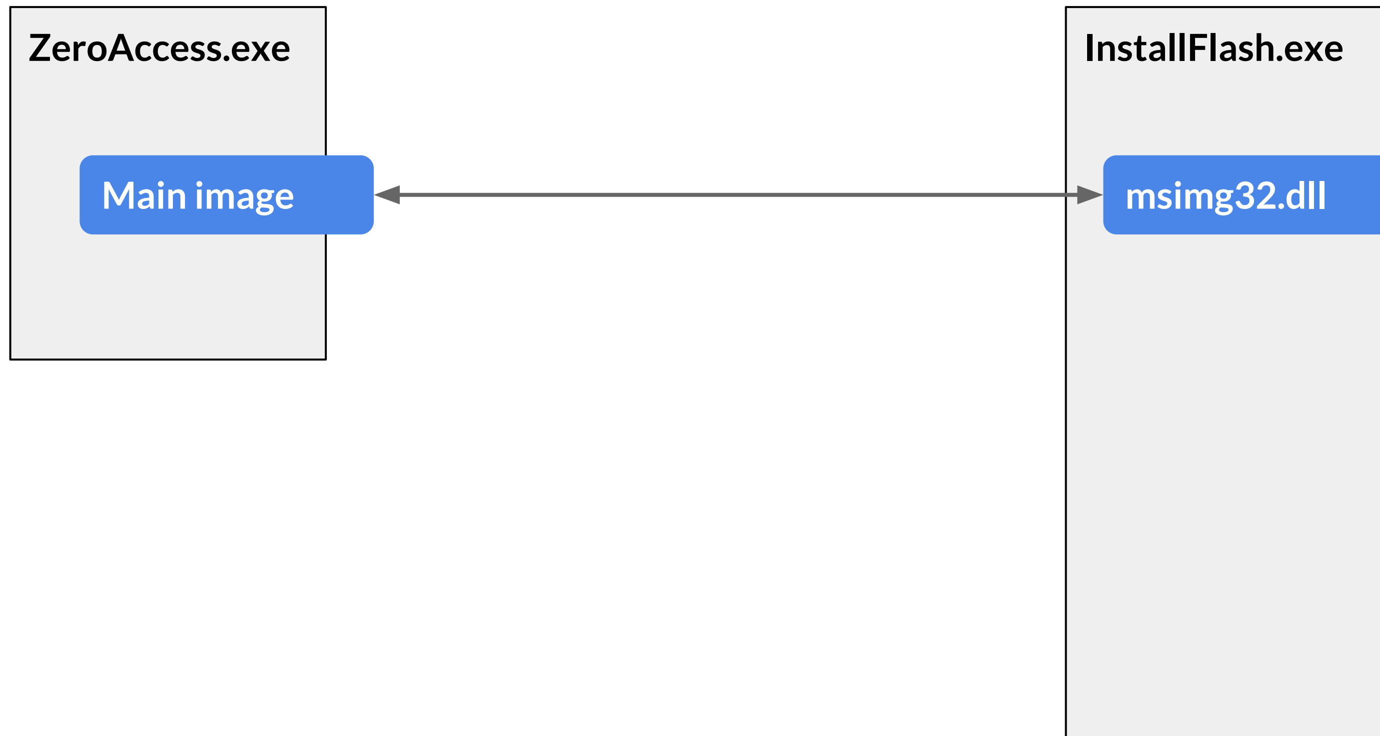
# Zeroaccess - Visualized timeline



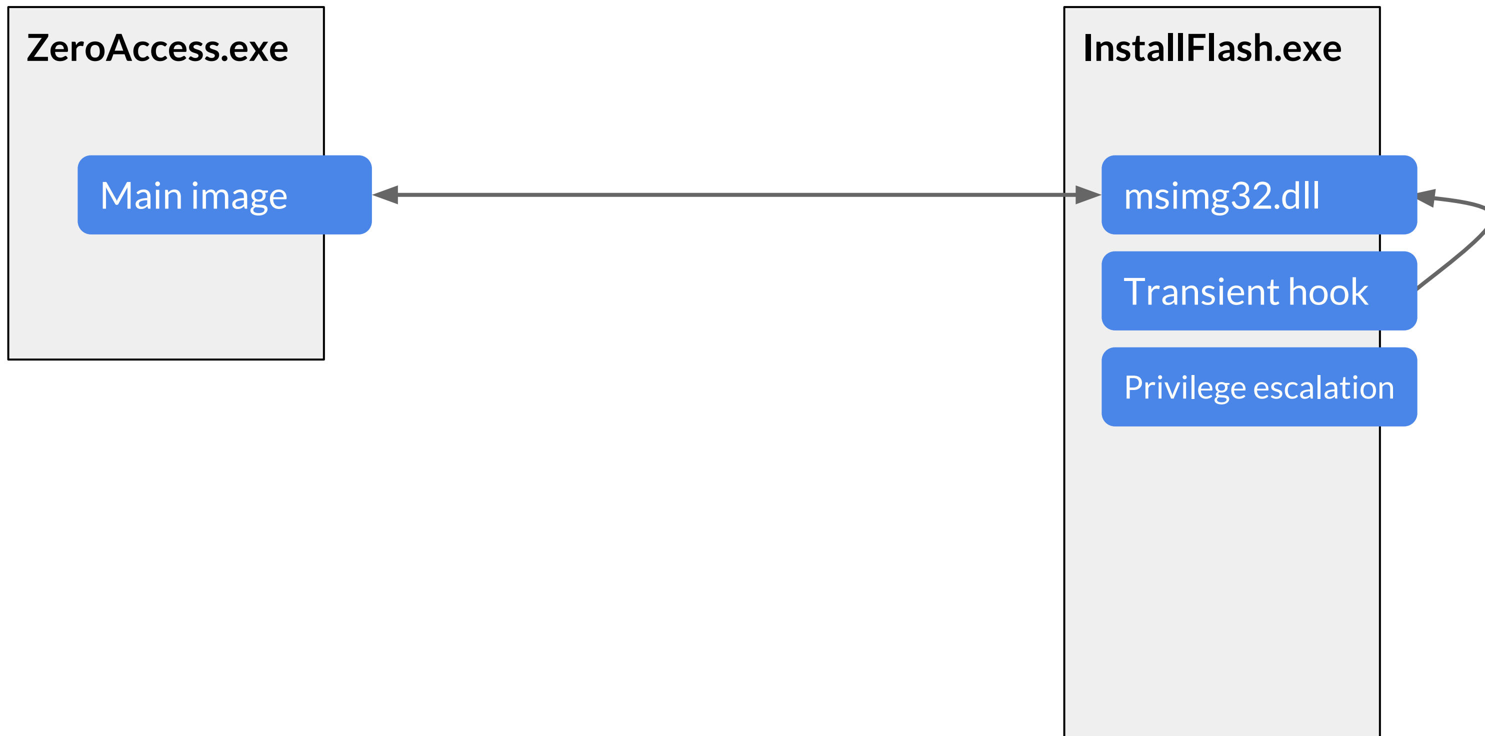

First analysis / hypothesis:

- `zeroaccess.exe`, `InstallFlashPl.exe`, and `services.exe` are malicious / corrupted
- `InstallFlashPl.exe` elevated privileges to inject into `services.exe`

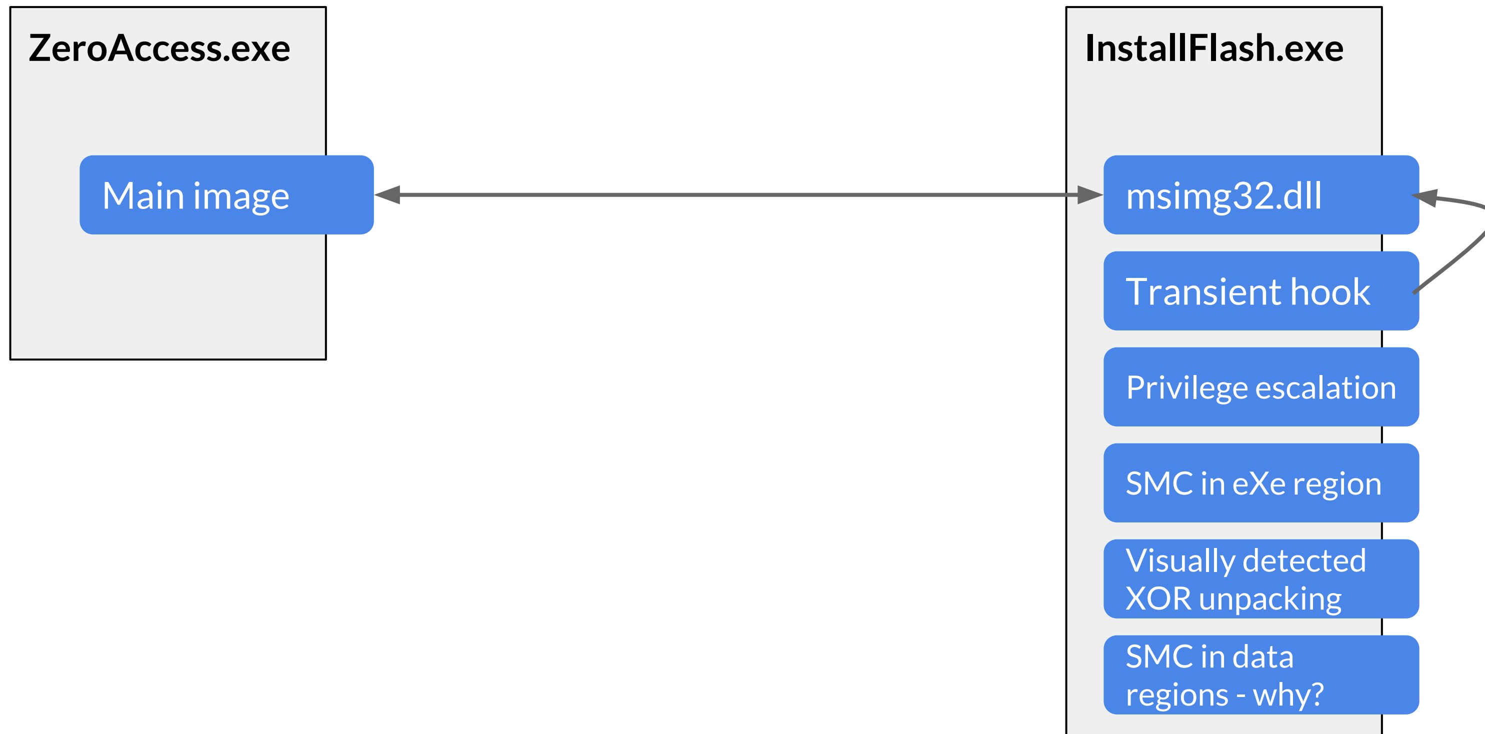
# Insights from video I



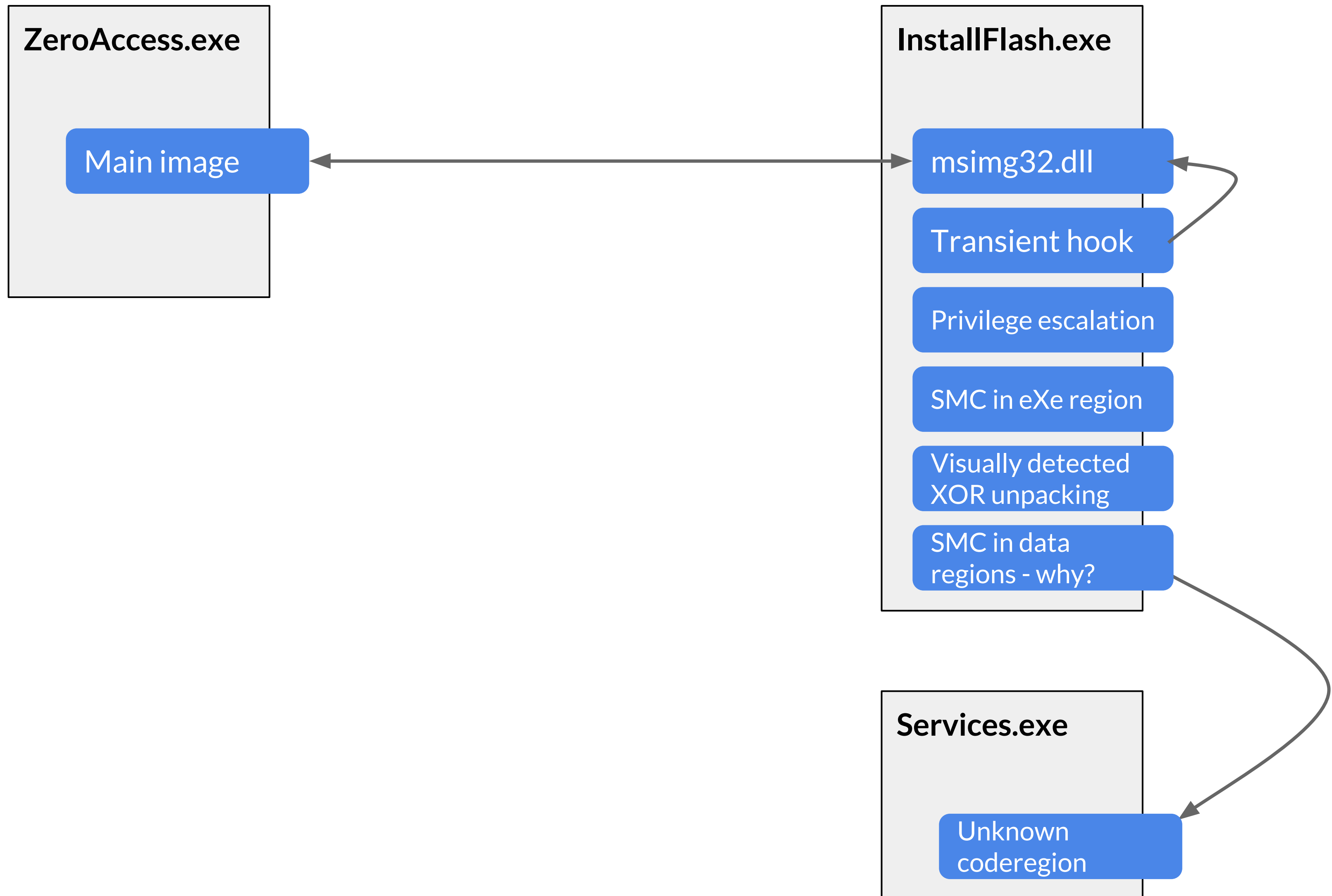
# Insights from video II



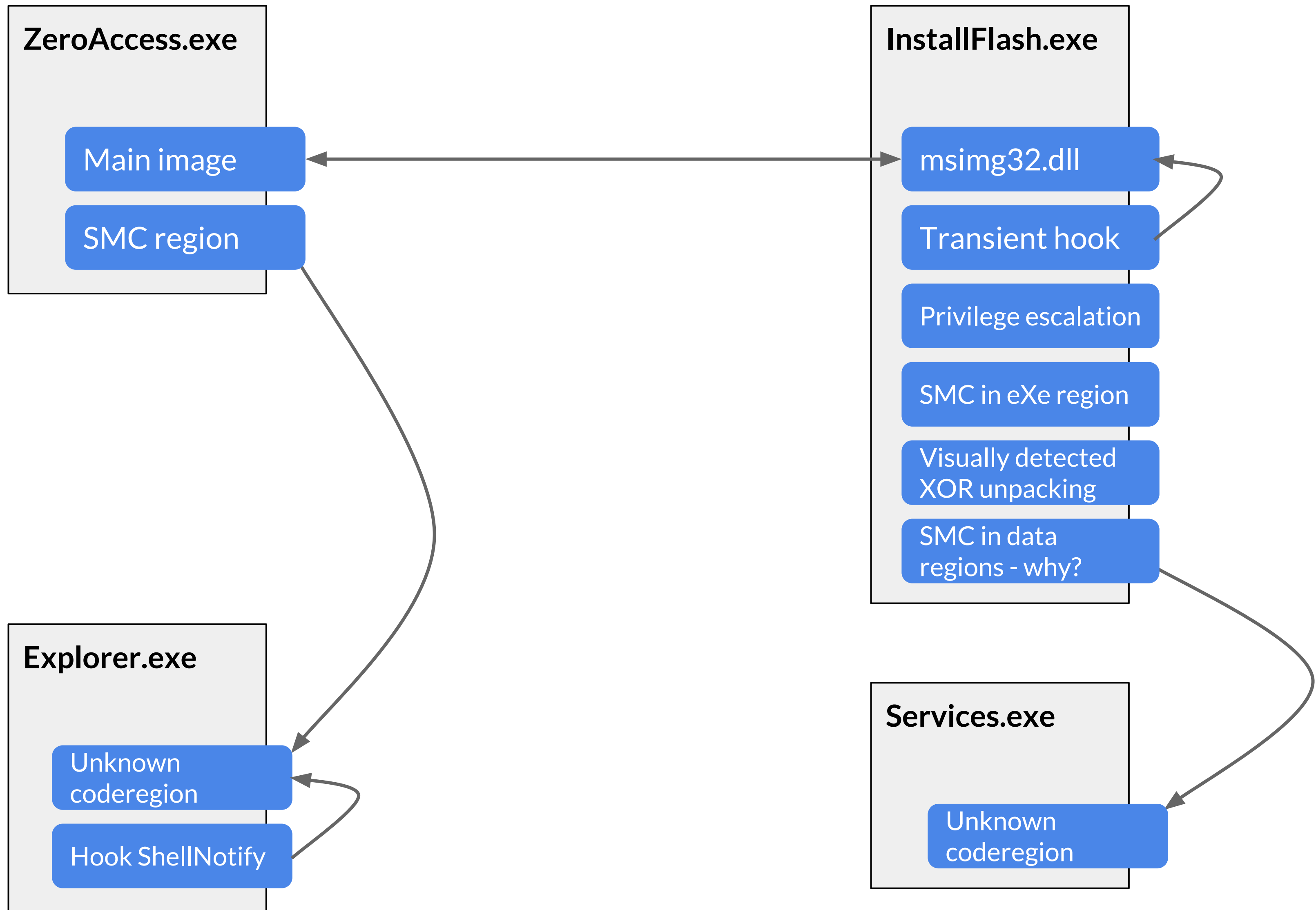
# Insights from video III



# Insights from video IV

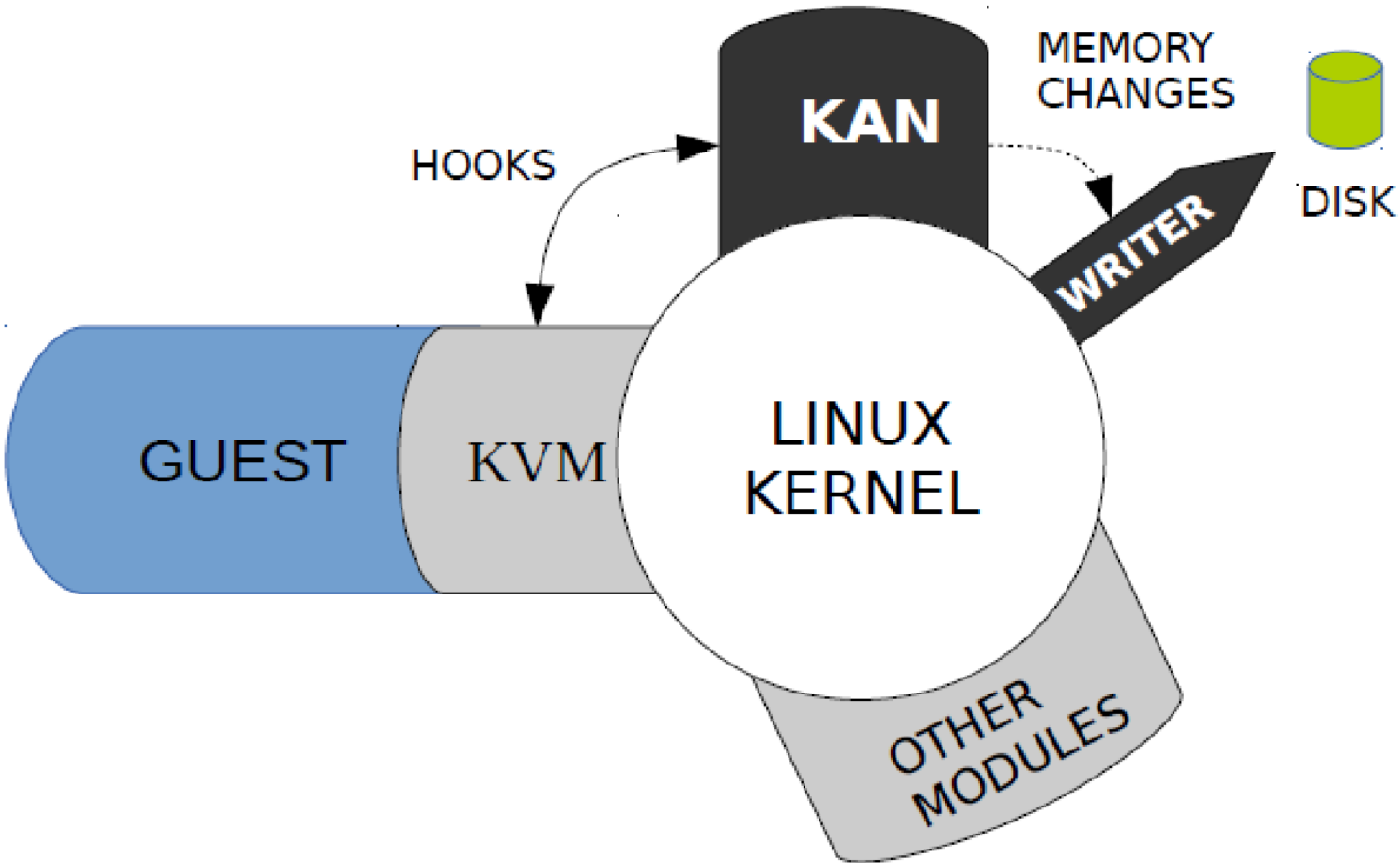


# Insights from video IV



# Memory trace acquisition

# Architecture memory tracing engine



# WHEN to make snapshots: Triggering

- Basically everything that leaves guest execution (VMEXIT)
- System call trigger
  - e.g. Snapshot after every `NtWriteVirtualMemory`
- Temporal trigger
  - e.g. Snapshot every 20ms
- Manual trigger
  - Single snapshot by `echo 1 > /proc/kan/single on the host`
- Guest trigger
  - Instrument a guest binary with CPUID / VMCALL instructions
- Choice of triggers matters

# “Generic” system call trigger

- Injects small agent into guest
    - Only leaves guest execution for configured system calls
    - Increases speed at cost of forensic neutrality

- Hooks
    - 32bit: SYSENTER
        - `MSR_IA32_SYSENTER_EIP` => set to agents address
    - 32bit: SYSEXIT
        - `MSR_IA32_SYSENTER_CS` => set to 0 => trap #GP
    - 64bit: SYSCALL
        - `MSR_LSTAR` => set to agents address
    - 64bit: SYSRET
        - RCX => set to non canonical address => trap #GP

- Forces a VMEXIT with VMCALL, CPUID or #GP

- Configurable
    - For each system call: Before and / or after system call

- Works on Linux & Windows: 32bit and 64bit

# HOW to make snapshots?

- When trigger a fires, memory needs to be dumped to disk. How?
  - Including meta data such as registers and timestamp
- Enumerate entire memory (guest physical memory)
  - Extended paging tables (EPT)
  - Dirty page tracking
- Write memory changes asynchronously
  - Copy in memory, async writing to disk
  - Increases write throughput up to factor 10x
- Limitations
  - Max. 512 MB guest memory
  - Max. one virtual CPU
  - Max. one VM running
- Requirements
  - Host CPU featuring EPT
  - Transparent huge pages (THP) disabled
    - echo never >/sys/kernel/mm/transparent_hugepage/enabled

# Memory tracing engine

- Performance
  - depends on triggering frequency
  - but system under acquisition can be used interactively
  - ~ < 30ms per snapshot on moderate hardware
- Operating system independent
  - Whatever runs under KVM is fine
  - In particular, Linux, Windows
  - 16bit / 32bit / 64bit
- "Relatively" stealthy
  - Minimal guest instrumentation (just the syscall trigger)
  - As stealthy as KVM

# Outlook & Conclusion

# Conclusions & outlook

- Memory traces can be useful
  - Can easily / quickly understand infection behavior
  - Guide analyst to interesting code regions for further manual analysis

- Forensic soundness in dynamic analysis
  - Can revisit memory trace any time, and examine the system state at capture time
  - Kind of hybrid between dynamic & static analysis

- Have seen rather simple analysis, can be done more:
  - "Completely" understand contents of memory traces and the significance of modifications
  - Correlate different memory traces

- Any questions, ideas… please get in touch here @recon or email bte1@bfh.ch

# Thank you for your attention!