malWASH: Washing malware to evade dynamic analysis

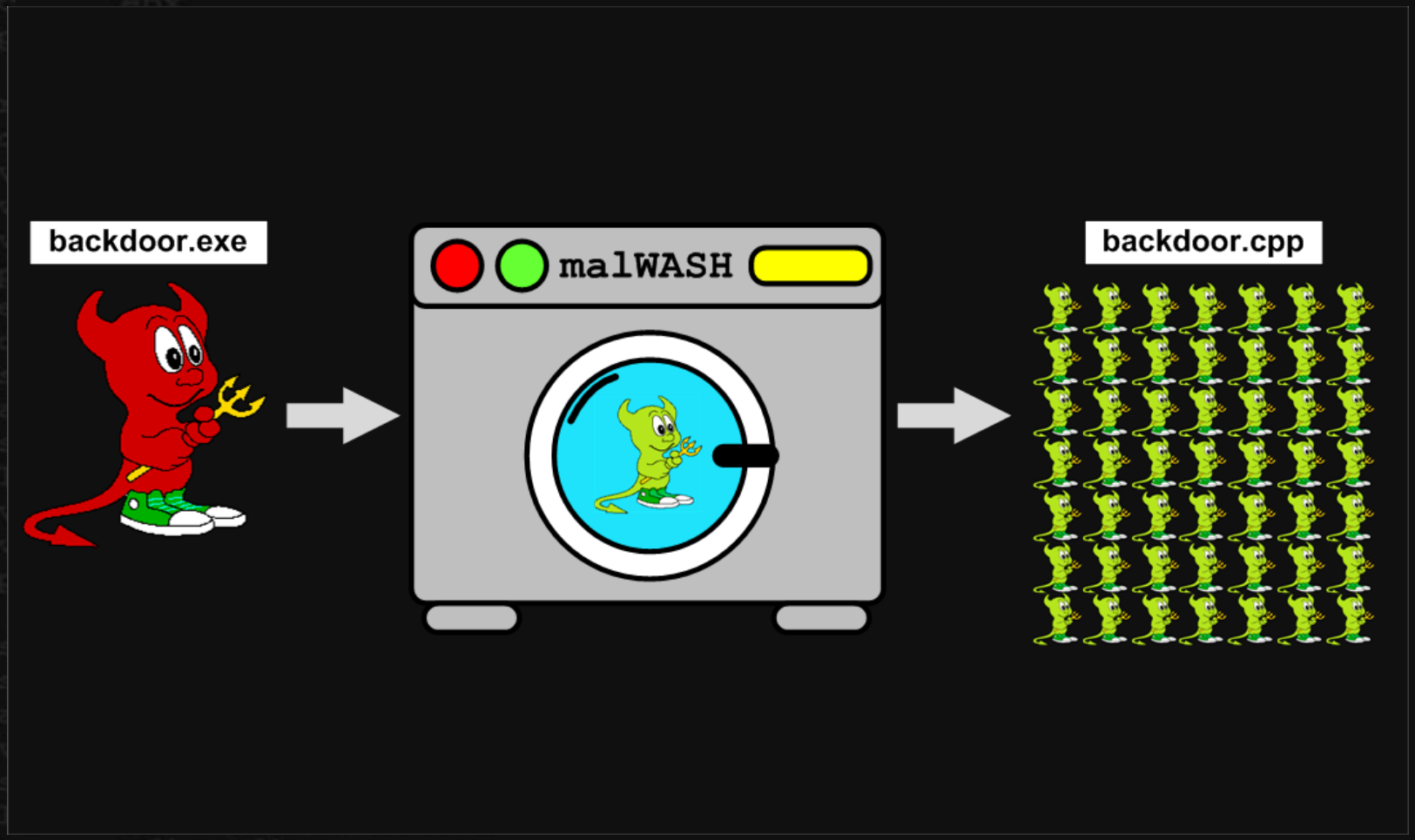
Kyriakos Ispoglou (ispo)
Mathias Payer

Computer Science Department,
Purdue University, West Lafayette, IN, USA

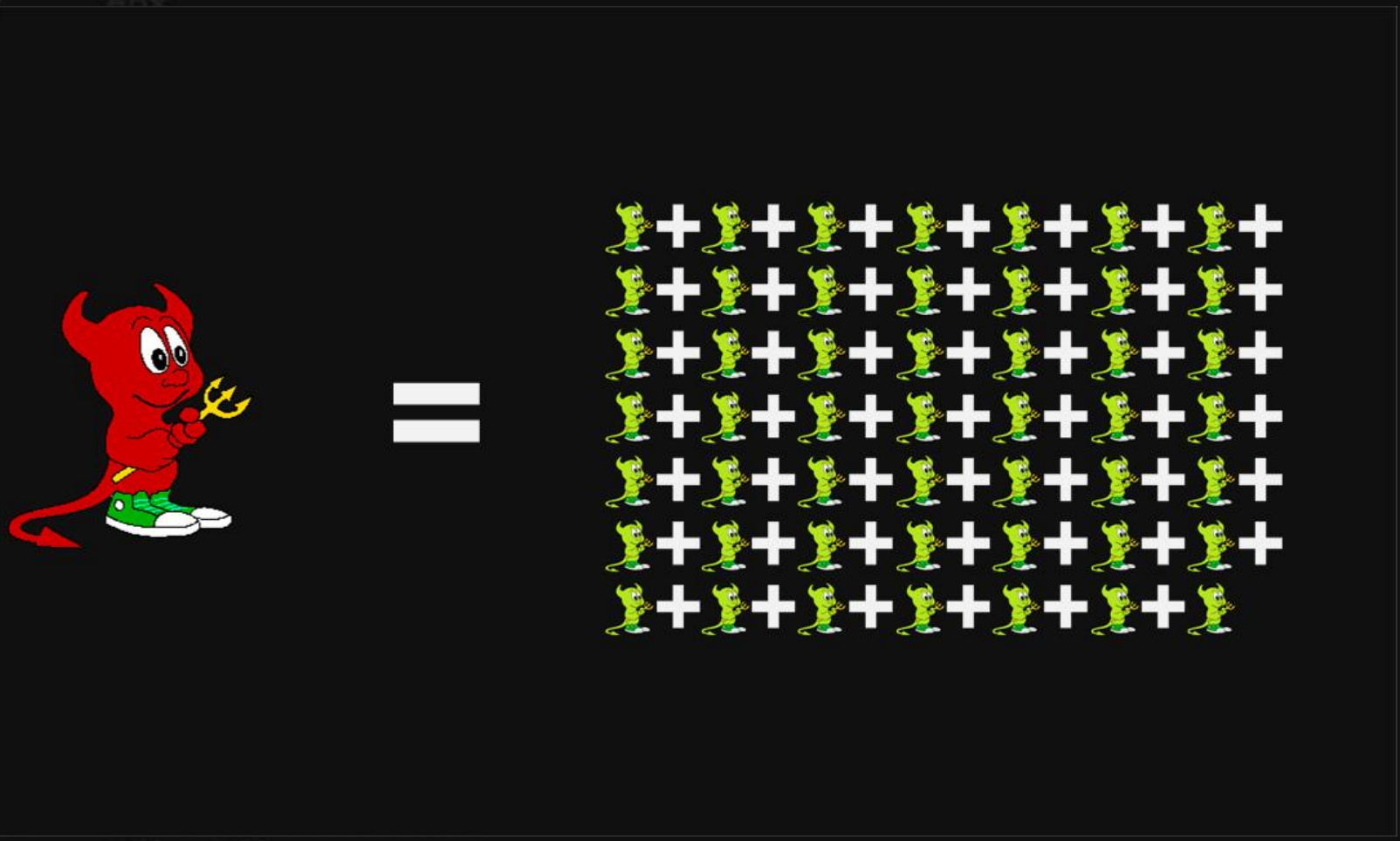
Motivation

- Malware must be stealthy
- Goal: Make existing malware undetectable
- Dynamic/Behavioral analysis is powerful
- Game over for attackers?

malWASH Concept



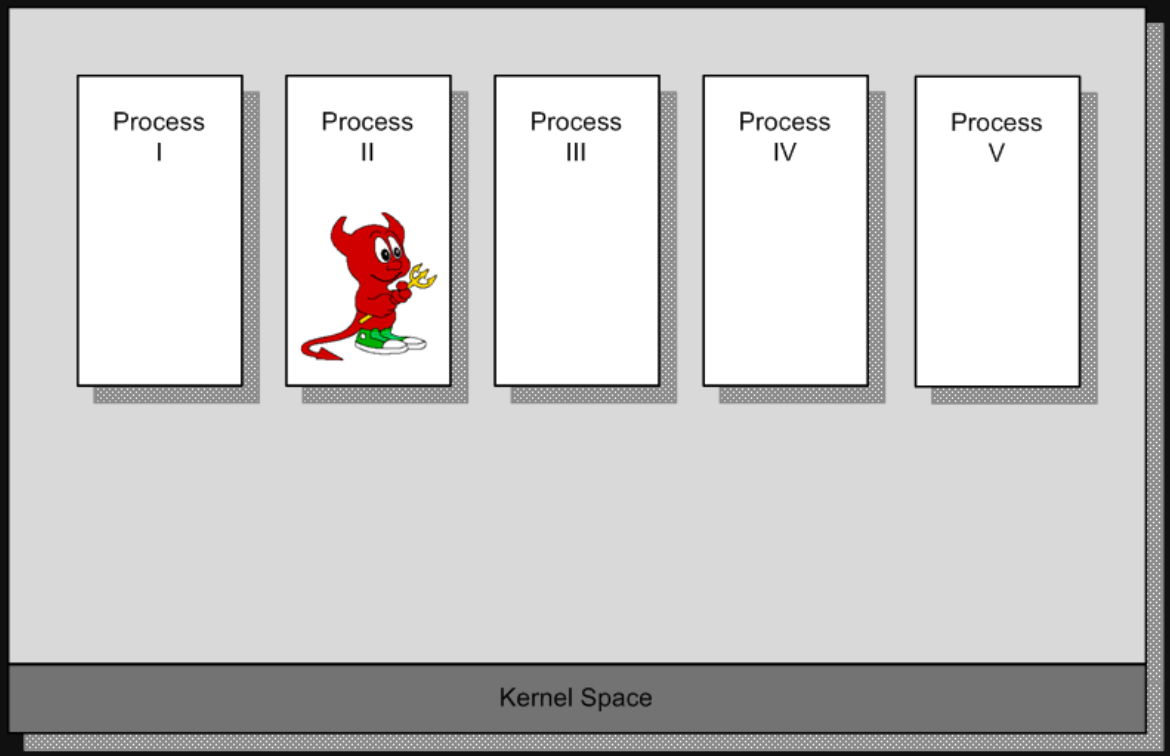
malWASH Concept



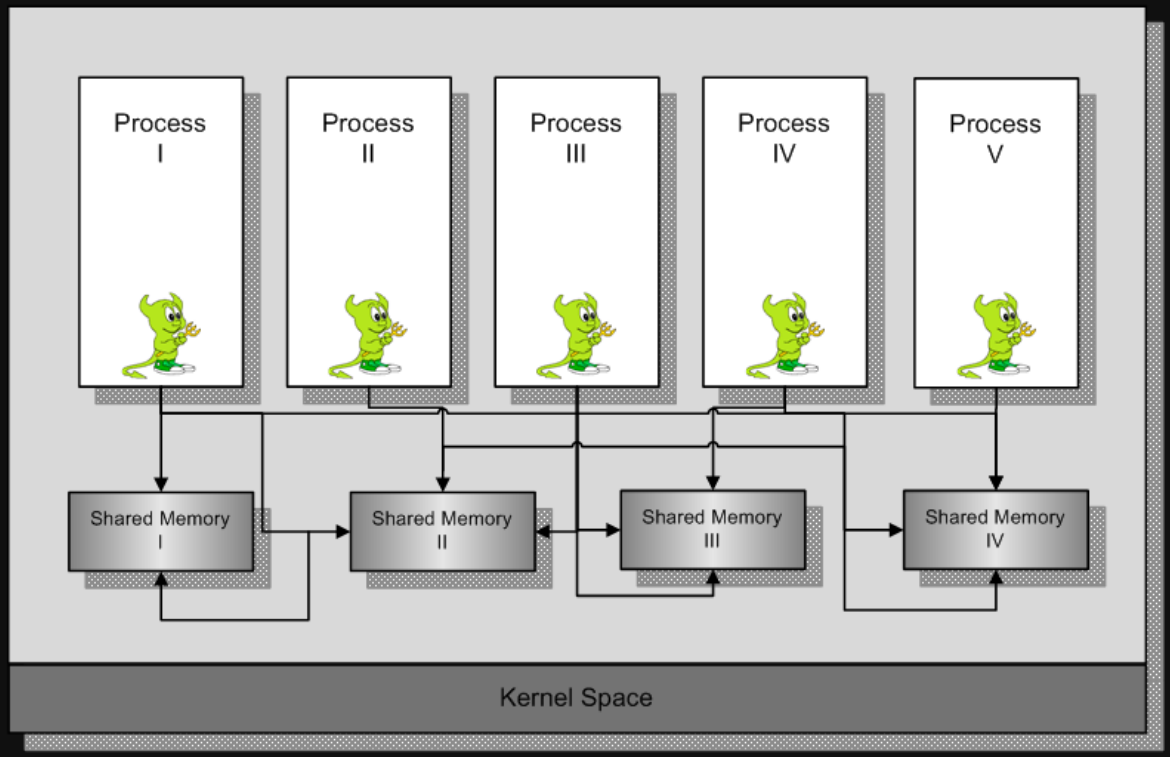
malWASH Concept

- Goal: Thwart behavioral and dynamic analysis
- Approach: Automatically distribute a program across a set of benign processes
- malWASH exploits the constrains of behavioral and dynamic analysis

Normal Infection



malWASH Infection



malWASH Idea

- Divide malware into hundreds of blocks
- Execute blocks in context of different processes
- Execution flow between all these blocks and original program is equal

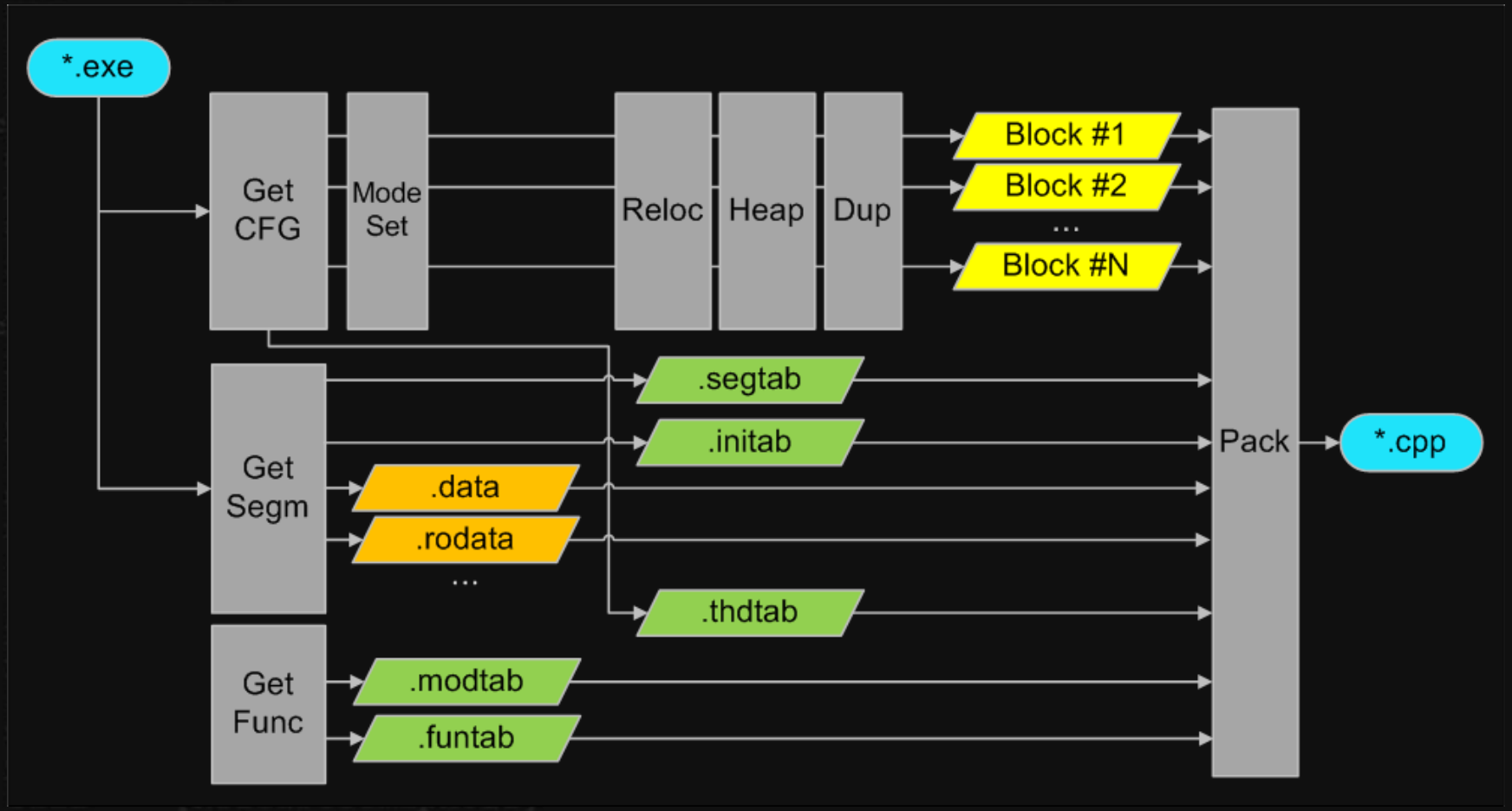
malWASH Design

- Emulator: Execute blocks inside another process
- Loader: Program that injects the emulators
- Distributed design
  - Resilience
  - Disinfection hardened

Implementation

- Consist of an offline and an online component
- Offline: Analyze binary and generate source file
- Online: loader + emulator

Offline processing



Offline processing

- Chop binary into small blocks and assign unique block identifiers (BID)
  - Granularity Mode: BBS, BAST, Paranoid
  - Policy: *"At the end of a block execution, ebx contains BID of next block"*
- Redirect control flow transfers to dispatcher
  - jmp, jcc (near/far jumps)
  - call, retn, retn XX
  - loop, loope, loopne
  - indirect jumps/calls

Control-flow translation: Example

```
retn
```

```
    xchg [esp], ebx
    cmp  ebx, $_RET_1
    jz   TARGET_1
    cmp  ebx, $_RET_2
    jz   TARGET_2
    ...
    mov  ebx, ffffffffh
    jmp  END
TARGET_1:
    mov ebx, $_BID_1
    jmp  END
TARGET_2:
    mov  ebx, $_BID_2
    jmp  END
    ...
```

Rewriting challenges

- Relocations are needed for:
  - memory accesses
  - function calls
  - heap requests
  - socket descriptors/HANDLES
  - threads
- After all, everything is packed in a C++ file

Example: Creating a descriptor

```
push edx
call ds:__imp__socket@12
mov [ebp+sock], eax
```

```
   push  edx
   nop
   jmp   DETOUR_ENTER
DETOUR_RETURN:
   mov   [ebp-sock], eax
   ...
DETOUR_ENTER:                          ; at the end of the block
   call  ds:__imp__socket@12
   call  near ptr $_CRT_DUP_SOCK   ; arg in eax
   jmp   DETOUR_RETURN
```

Example: Using a descriptor

```
push ecx
call ds:__imp__RegSetKeyValue@24
cmp  eax, 0FFFFFFFFh
```

```
    push  ecx
    nop
    call  DETOUR_PROC
    cmp   eax, 0FFFFFFFFh
    ...
DETOUR_PROC:                        ; at the end of the block
    mov   eax, [esp + 0x4]          ; HKEY hKey
    xchg  ebx, [esp + 0xc]          ; when >1 descr. are used
    call  near ptr $_LOC_DUP_DSC
    mov   [esp + 0x4], eax          ; replace them
    xchg  ebx, [esp + 0xc]
    jmp   ds:__imp__RegSetKeyValue@24
```

Online Component

- Loader selects processes, injects emulator
- Emulators start executing program
- Emulators coordinate program execution

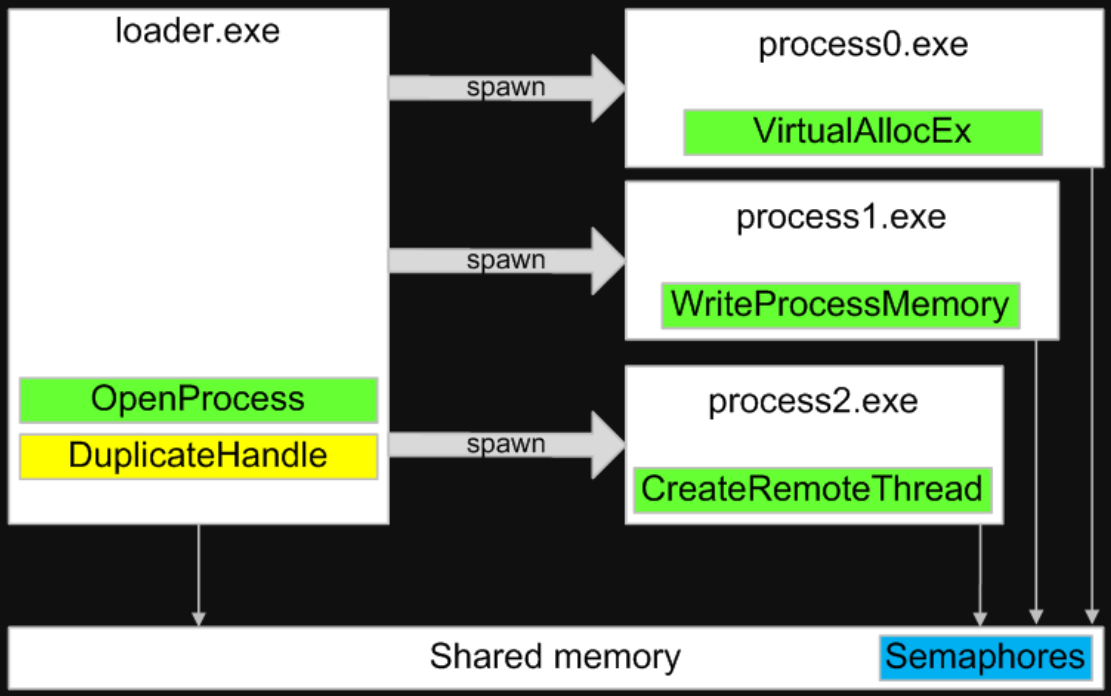
Online Component

- Process injection involves:
  - OpenProcess()
  - VirtualAllocEx()
  - WriteProcessMemory()
  - CreateRemoteThread()
- A noisy operation

Online Component

- Mitigate detection by using NT API functions:
  - ZwOpenProcess()
  - ZwAllocateVirtualMemory()
  - ZwWriteVirtualMemory()
  - NtCreateThreadEx()
- And/or, recursively use of the malWASH concept to split the loader process

Mitigating loader detection



Online component: Emulator

- Written in ~5,500 Assembly LoC
  - Only 14kB in size
- Emulates memory accesses and function calls
- Coordinate shared execution environment

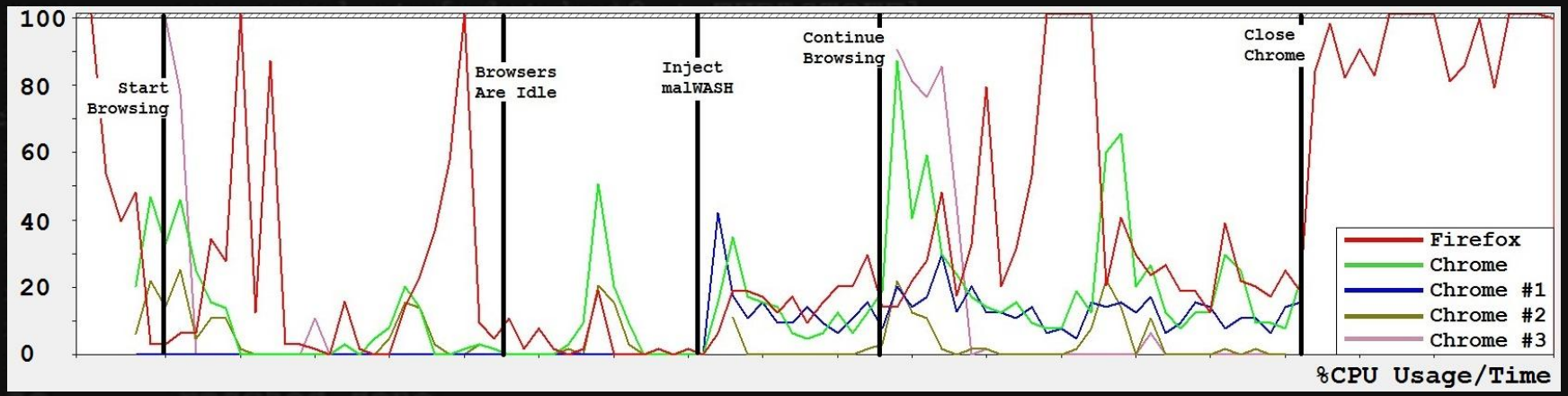
Online component: Emulator

- Emulator has more features:
  - Process mailboxes
  - Duplication Table (duptab)
  - FILE* replacements
  - Heap manipulation
  - Other replacements (e.g. ExitProcess)
  - Call Cache
  - Scheduler (for multi-threading)
  - Recovering killed emulators

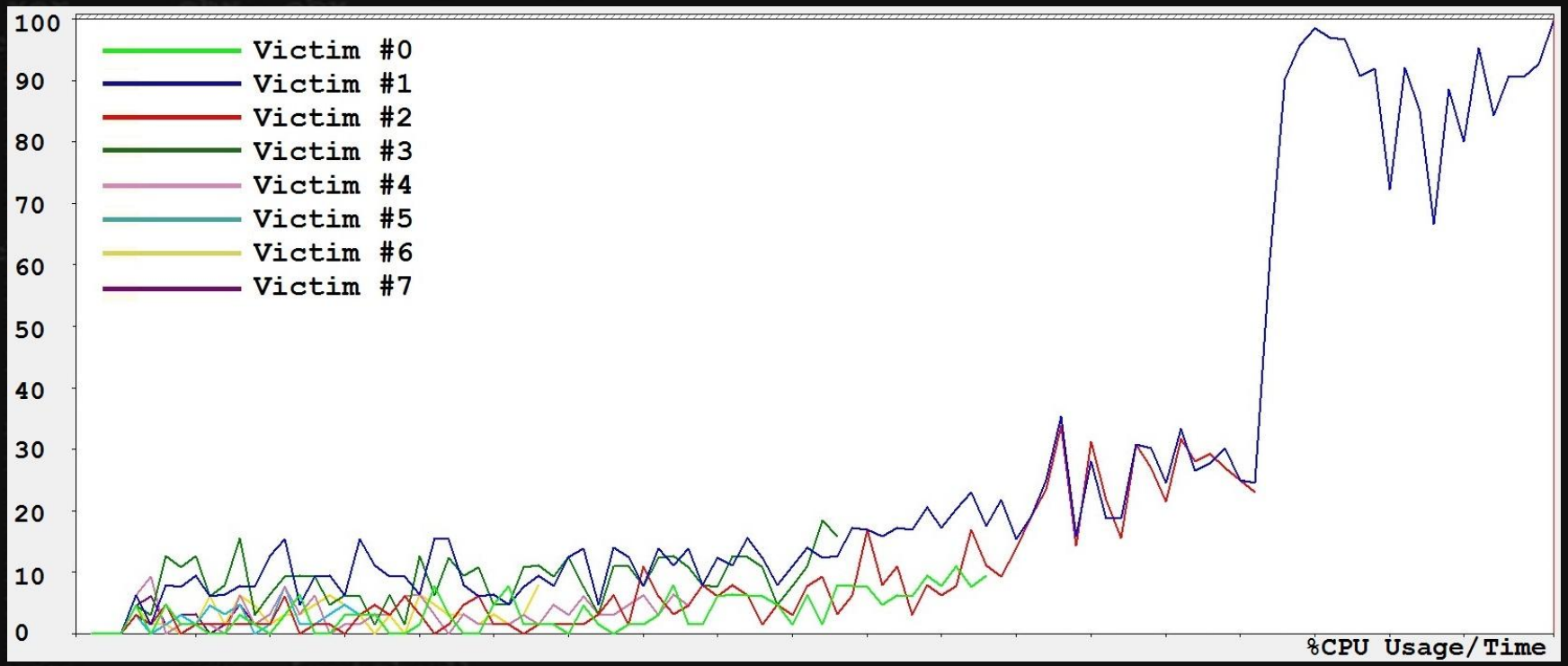
Evaluation

- malWASH evaluated with 8 malware samples
- Each sample was split in all 3 modes
- Google Chrome selected as victim process
- Successful injection in 1, 2, 4, and 8 processes

Evaluation: %CPU Usage



Evaluation: Resilience



Defeating malWASH

- Detecting malWASH programs is hard
- Goal: Detecting malWASH itself
- Identity of original program gets protected

Defeating malWASH

- malWASH leaves detectable execution traces
  - Honey pot processes, shared memory correlation, abnormal process overhead, behavioral process discrepancies, loader detection (use of pre-infected processes), emulator detection, ...
- All of them can be mitigated
- malWASH can be used to protect itself!
  - Distribute the emulator thread among a set of threads

Conclusion

- malWASH distributes program execution among benign processes
- Detection using dynamic analysis is challenging
- A good detection mechanism must:
  - Detect malWASH, and
  - Not detect anything else

Questions?

- malWASH source code:
  https://github.com/HexHive/malWASH
- Contact Information:
  - *ispo@purdue.edu*
  - *mathias.payer@nebelwelt.net*
- *Github pages:*
  - https://github.com/ispoleet (ispo)
  - https://github.com/gannimo (Mathias)