malWASH: Washing malware to evade dynamic analysis

Kyriakos K. Ispoglou
Purdue University

Mathias Payer
Purdue University

Abstract

Hiding malware processes from fingerprinting is challenging. Current techniques like metamorphic algorithms and diversity generate different instances of a program, protecting it against static detection. Unfortunately, all existing techniques are prone to detection through behavioral analysis – a runtime analysis that records behavior (e.g., through system call invocations), and can detect executing diversified programs like malware.

We present malWASH, a dynamic diversification engine that executes an arbitrary program without being detected by dynamic analysis tools. Target programs are chopped into small components that are then executed in the context of other processes, hiding the behavior of the original program in a stream of benign behavior of a large number of processes. A scheduler connects these components and transfers state between the different processes. The execution of the benign processes is not impacted. Furthermore, malWASH ensures that the executing program remains persistent, complicating the removal process.

1 Introduction

Malware (and fighting malware) is an important aspect of computer security. Malware by itself does not exploit security vulnerabilities but is the payload that is executed post-exploitation. Consequently, malware is only successful if it is stealthy and remains undetected. Sophisticated, undetectable malware is therefore a required asset for attackers. Anti Virus systems (AV) are based on signature detection and static analysis. Although this method has limitations, it is well-proven, reliable, and accurate. The AV identifies malware by looking for known patterns or characteristics. Due to its simplicity and accuracy, signature-based detection remains widely used.

Malware authors bypass signature-based detection by using metamorphic [33] algorithms and diversity. These techniques generate instances of the same binary that have different signatures, while maintaining the functionality of the binary. Defenders quickly realized that all generated instances have the same functionality, and started to identify the behavior of the malware instead of the signature [2]. Dynamic analysis executes the malware to reveal its behavior. This method is simple but effective, e.g., a typical keylogger repeatedly performs a sequence of specific system calls. No matter how obfuscated the binary is, these system calls are repeated in the same order, making the keylogger easily detectable.

A simple technique to bypass behavior based detection would be to insert bogus system calls (i.e., system calls that do not affect the original execution) between real ones. An analysis can likely filter out bogus system calls, thereby mitigating this naive technique. We propose a sophisticated, novel mechanism to hide malware from behavior-based analysis. Rather than executing the program in a single process, we automatically distribute the program across a set of pre-existing, benign processes. Our approach is based on a simple observation: although we cannot modify the executing system calls and their order of execution in a binary, we can hide them within the stream of system calls that are performed on the *entire* system.

To spread our system calls across the stream of calls for the *entire* system we propose injecting our system calls into a set of existing processes on the system. To do this, the original binary is chopped into small chunks. Each individual chunk only contains limited functionality and therefore executes few system calls. These small chunks and an "*emulator*" are then injected into multiple running processes and blend into the stream of executed system calls. Each emulator then selects the individual chunks to run, captures state, and coordinates with the other emulators who continues execution.

Detection tools that observe behavior based on a per-

process analysis no longer see the complete sequence of system calls that the program executes. Each injected system call is hidden in a set of benign system calls and the program functionality is spread across a set of benign processes, executing benign code (in addition to the injected one). Tracking the system calls of all applications globally and trying to look for malicious patterns is a strictly harder problem, as system calls from the injected binary are spread out in the stream of system calls for the entire system. Consequently, methods like [14] which search for short sequences of malicious system calls fail.

Prior obfuscation techniques such as [32, 10] guarantee that the actual computation remains the same, which is a required, fundamental property that enabled behavioral analysis. malWASH guarantees equal functionality, while bypassing behavioral analysis. The design of our "malware" engine allows chopping and executing arbitrary programs. To keep our Windows-based prototype implementation general but simple, we constrain the execution environment, and assume that the binary has some specific properties (defined in Section 4). We evaluate our malWASH prototype implementation with samples from different malware categories and show that our implementation successfully chops and executes the programs.

Beyond stealthiness, malWASH offers another interesting property: *resilience*. The malware is distributed as it is injected into multiple benign processes and executes as part of them. Therefore, killing a single process does not stop the execution of the malware as it can reinstantiate itself from any remaining emulator. The only way to stop malWASH is to kill *all* infected processes at the same time, before any process reinfects a new process.

The contributions of malWASH are:

- Design of an execution engine that thwarts behavioral and dynamic analysis.
- Creation of fully persistent malware that continues executing as long as at least one emulator remains.

Furthermore, the design of malWASH, has some very interesting properties. First, even if malWASH is detected, the actual binary remains obfuscated in a plethora of processes, complicating reverse engineering. An analyst would first have to correctly reassemble the binary. Second, all of the existing obfuscation and diversity techniques can be used with malWASH.

2 Background and Related Work

Over the last decade, many techniques have been proposed to enable obfuscation and diversity, with the goal of hiding malware from AV systems. One of the oldest methods to detect whether a given binary is malicious or not is to use static analysis detection [30, 27]. Anti-disassembly mechanisms [10, 8, 1] allow malware authors to bypass static analysis and companies to protect their IP against, e.g., illegal distribution. Although powerful, anti-disassembly techniques are infamous; benign programs have no reason to obfuscate their code as obfuscation may impact performance, stability, and the ability to reproduce crashes. Even though analysis of binaries protected by anti-disassembly is hard, it is straight-forward to check whether such protections were applied, e.g., detecting an encrypted PE header [6]. An AV can exploit this fact and flag a binary as malicious without trying to analyze it, as using such obfuscation is a strong indication that the binary is actually malicious.

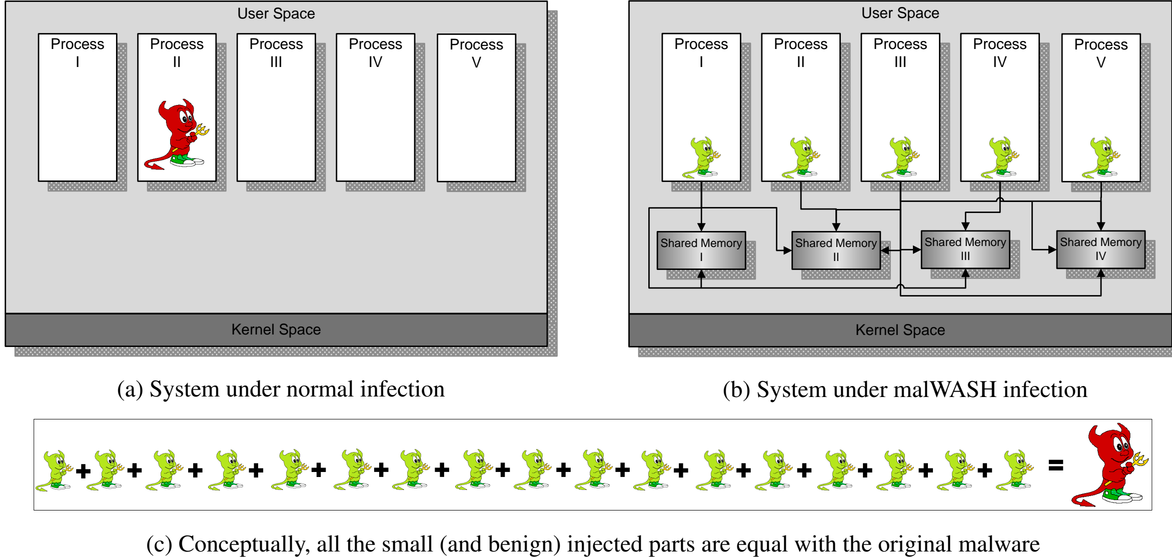
Furthermore, these mechanisms have to eventually *reveal* their payloads and execute it. Techniques like dynamic analysis and sandboxes, analyze the malware and compare the behavior against well-known patterns. Anti-debugging techniques [3, 32] along with VM-detection [11] are used to change a program's behavior when a sandbox or a debugger is detected. All these methods share that the actual execution of the malware, when not being debugged, remains constant (this is a guaranteed property). Consequently, observing the behavior of the executing malware always yields the same observation (e.g., same system calls in the same order).

Improved obfuscation mechanisms were proposed, notably using Return Oriented Programming (ROP) to hide a malware within a benign program [21, 25]. Although effective, a ROP-style execution can easily be detected [23, 24, 5, 16, 4, 22]. Another interesting obfuscation technique is movfuscator [7], which compiles a program using only `mov` instructions. This makes analysis extremely hard, but detecting that movfuscator is applied to a given binary is trivial. Any use of movfuscator is an indication that a binary is malicious, even if there is no information on what the binary does.

The concept behind the previous approaches, was to *hide* a malicious payload within a program. Another approach is to "get rid" of the malicious payload, by forcing another program to execute it for you. Metasploit's meterpreter [20] uses DLL and Reflective DLL [12] injection, to inject a malicious payload into another process's address space.

The common property of all the aforementioned protections is that any malicious action happens within the context of a single process. Here is where dynamic analysis [9] takes place. This is a powerful method that tries to classify a program or a process as malicious by observing its behavior (e.g., system calls, involved files, or network connections). Using dynamic analysis, it is possible to detect new, unknown malware just by matching the behavior of the binary.

Many dynamic analysis methods have been proposed



(a) System under normal infection

(b) System under malWASH infection

(c) Conceptually, all the small (and benign) injected parts are equal with the original malware

Figure 1: A comparison between normal infection and malWASH

to detect malware. Methods based on execution tracing [19, 14, 17, 15, 31, 2], inspect executing traces, looking for malicious patterns of system calls. However, when a binary runs under malWASH, is significantly complicated due to the distributed nature of our approach: (i) the execution trace of a process, contains only a small and out-of-order subset of system calls, and (ii) any sequence of system calls of the original binary is distributed among multiple processes, because each quantum given to malWASH, contains only a few system calls (e.g., 1 or 2).

The most recent malware detection methods use machine learning techniques to classify a binary as malicious or not [13, 26, 18, 28, 34]. However, these methods all assume that the malware runs in a single process and that only malicious system calls are executed by a process.

Even though the original binary is well hidden and protected, defenders could try to detect the malWASH emulator itself and not the binary it emulates. However, the idea of malWASH can also be used to protect malWASH itself. As we show in Section 5.3, the use of sub-emulators (small emulators that emulate the original emulator) along with other hardening methods in emulator, makes detection challenging.

3 malWASH Design

The design of malWASH follows a simple concept: breaking a program into small pieces and hiding these pieces in benign processes (see Figure 1). Conceptually, malWASH works as an emulator that (i) executes individual instructions of the program and (ii) coordinates with the other active emulators to create a correct flow of execution.

Behavioral malware detection is carried out per process (or per thread). After analysis, an *individual* process can be flagged as malicious. We believe that scaling behavioral analysis to a group of processes or threads is hard due to the exponential explosion of possible combinations of system calls across processes. malWASH introduces an emulator that allows the execution of a target program in a set of host processes. In the most stealthy mode of malWASH, a host process executes a single instruction of the emulated program per time slice. Detecting this one instruction within the millions of instructions that get executed by the process is highly unlikely.

malWASH takes as input a binary file and produces a C++ source file that embeds all the required parts of the binary along with all malWASH components. Using a source file as output enables further binary obfuscation processes (e.g., metamorphism). This means that all the existing protection methods against static analysis and signature detection work on top of malWASH.

malWASH operates in two phases. In the first phase, the original program binary is "chopped" into hundreds of small pieces and all the required information is extracted from the binary (segments, loaded libraries, relocations, global data and thread information). All these components (including those from malWASH) are encoded as character arrays and packed into a single C++ source file.

Chopping the binary into components is challenging as control-flow transfer instructions (e.g., `jcc`, `jmp`, `call`, and `ret`) may transfer control of the execution to a point that is not in the current address space. Therefore,

the initial chopping is done at the basic block level. This way we know that only the last instruction transfers control to other locations. Using an emulator lookup function, we can replace the original instruction with a set of instructions that recover control-flow (possibly signaling another process to continue). Once we finish chopping on basic blocks, we can further chop the basic blocks themselves into new, smaller blocks, or start coalescing basic blocks to larger blocks.

Splitting opens a trade-off between *efficiency* and *stealthiness*. Using smaller blocks, malware signatures disappear and dynamic analysis detection tools fail to observe malicious behavior in the block. On the other hand, because there is a lot of overhead to transition between executed blocks (capturing state, selecting the next block to execute, and scheduling which process should execute the next block), fewer blocks will lead to less overhead from the emulator.

Once the source file is compiled, the program is ready to execute. The second phase of malWASH takes place when it starts execution. The first component is the loader, which looks for a set of "suitable" processes. The amount of emulators used is flexible and user-configurable. A good candidate is, e.g., the Google Chrome browser as it spawns many communicating processes that are perfect candidates for injection. A process is suitable when it allows another process to inject and execute code into its address space. Obviously, these instances need to cooperate, so a stealthy, reliable communication channel is needed. For this reason the loader also initializes a small set of shared memory regions for use by all the malWASH emulators. These shared regions contain data segments, stack, heap and all metadata that emulators are needed in order to cooperate and execute the blocks. Instead of shared memory, other communication mechanisms can be used such as pipes, files, network ports, or even covert channels.

Using shared memory regions has several advantages over process messages: (i) message may get lost and (ii) someone may observe messages between processes that are irrelevant to each other. If the emulator from process A communicates with the emulator from process B, it writes to the shared region that emulator of process B is waiting to read. Someone may still observe that there are new shared regions between processes, but as we show in Section 5.3, this information is of limited use.

Emulation of the malware begins after the loader terminates. Control is transferred to the first emulator (there is no central scheduling emulator) which executes its basic block, and then transfers control to the next emulator. At any time, exactly one emulator runs a piece of the original program (except for multi-threading programs where multiple emulators can execute different blocks of the program as long these blocks belong to different threads). Semaphores and Mutexes synchronize the emulators and ensure that no more than two processes will execute blocks from the same thread at each time.

When an emulator successfully takes the semaphore, it executes the next block of the malware. Before executing the next block, a context switch is performed and all memory accesses and imported function addresses are properly relocated. During the execution, current instructions within a block are executed transparently, without knowing of the emulation. After the block is executed, a context switch is performed, saving the current state of the program in the shared region and the emulators will coordinate to find which one will execute the next block. Note that different scheduling policies can be implemented to select which emulator executes the next block, we use a simple race. This distribution of emulators results in an address space independent execution.

When a process that contains an emulator terminates, the other emulators can continue the execution and the malware will continue to execute. The other emulators can detect the missing component and invoke the loader to reinitialize the missing emulator in a new process, keeping the total number of emulators constant. This means that as long as there is at least one emulator running it can recover from killed instances. Removing or stopping the malware requires that all emulators must be killed at the same time. The emulators run exclusively in memory, making it harder to detect as there are no persistent files.

4 Implementation

malWASH takes a binary program and distributes its execution across a set of benign processes, coordinating the global state of the program and the scheduling between the individual components. In the most fine-grained configuration, each instruction of the target program is a different entity. The Windows-based implementation of malWASH draws ideas from several areas: binary analysis to chop the program into individual components, binary translation to manipulate the control execution of each block, to coordinate between individual blocks and to orchestrate scheduling, and snapshotting to capture and synchronize program state across the different processes.

Our malWASH prototype implementation (available at `https://github.com/HexHive/malWASH`) consists of an offline and an online component. The offline component runs the binary analysis, chops the program into individual components, and prepares the emulator. The online component includes the loader that injects components into different processes and the emulator which orchestrates and coordinates the execution of the program among all the different host processes.

Property	Prototype Implementation	Design
Obfuscated	No	Depends
Self Modifying	No	Yes
Polymorphic / Metamorphic	No	Yes
Packed	No	Yes
Anti disassembly	No	Yes
Anti debugging	(Yes)	(Yes)
Non PIE	Depends	Yes
Use Heap	Yes	Yes
Multi Threading	Yes	Yes
W+X sections	No	Yes
Non x86	No	Yes
Statically linked	Depends	Depends

Table 1: List of supported properties by design and implemented in the current prototype.

By extracting the components offline, we can fall back on existing tools for the underlying binary analysis and, more importantly, our emulator does not require disassembly functionality. To keep the implementation prototype simple, we have restricted the (implemented) functionality of the emulator. Our emulator supports the full x86 instruction set (with a special focus of the control transfer instructions). Anti debugging features of the original binary can be mitigated by our translation and analysis process. The current implementation does not support x86-64 code and obfuscated or any form of self-modifying code (a design and engineering decision as otherwise the emulator would require its own binary analysis framework and disassembly functionality, vastly increasing the size of the emulator). Table 1 highlights the design trade-offs.

4.1 Phase 1: Chopping the binary

malWASH uses an IDA pro plugin to "chop" the binary. If IDA fails to analyse the binary, our tool will fail as well. Our plugin uses a Depth First Search (DFS) to disassemble the program from its entry point. This disassembly phase recursively follows control-flow transfer instructions and thereby recovers the Control-Flow Graph (CFG) of the binary, assigning a Block IDentifier (BID) to each basic block. These initial basic blocks can further be chopped into smaller pieces, depending on the configuration:

BBS (Basic Block Split) mode: the basic blocks are used as is.

BAST (Below AV Signature Threshold) mode: basic blocks are chopped so that each block is below a configurable threshold (we used 16 bytes for our experiments).

Paranoid mode: basic blocks are chopped to include only a single instruction.

4.1.1 Control-Flow Transfers

After binary analysis, each basic block ends with a control flow transfer instruction (e.g., jcc, jmp, call, or return and their variants). In BAST or Paranoid modes we have to insert additional transfer instructions to connect the newly chopped basic blocks. These instructions are replaced with a set of instructions that execute a lookup of the target block, transferring execution to another process if necessary. By convention, our binary analysis rewrites the basic block so that the target BID is in the `ebx` register (spilling the register if necessary). This is not optimal from a binary translation perspective but the context switching overhead to another process will dominate overhead and lookup efficiency is not a key concern.

Indirect control-flow transfer instructions like indirect jumps, indirect calls, or return instructions are harder to handle as the target BID is usually not statically known. For switch statements, IDA Pro can often recover the actual targets and replace them with the corresponding BIDs. For all remaining indirect control-flow transfer instructions we have to execute an online lookup that translates a target address to a BID. This lookup can use a table of all target locations, or, e.g., in the case of return instructions, we can use the CFG to identify all possible call sites and encode the return targets directly in the code as follows (see Figure 2).

These replacements ensure that the control flow transfers are translated correctly and allow the emulator to keep executing the target binary. Any calls back to the emulator request a new target in `ebx` and dispatch to the next block.

4.1.2 Block relocations

All external references within a block must be relocated at runtime. External references can either be functions from imported modules or constant references to segments (e.g., *data*, or *rdata*). Our block metadata keeps the offset of the addresses that need runtime relocation, according with the type of relocation. In cases of indexed array accesses, or constant pointers that point to constant addresses, all we have to do is to relocate the base address.

```
; if retn is used
    xchg [esp], ebx      ; backup ebx

; if retn NN is used
    mov  [esp+ARG], ebx  ; retn NN, ARG = NN*4
    mov  ebx, [esp]      ; get return address

; code for both cases
    cmp  ebx, $_RET_1
    jz   TARGET_1
    cmp  ebx, $_RET_2
    jz   TARGET_2
    ...
    mov  ebx, ffffffffh  ; ERROR
    jmp  END
TARGET_1:
    mov  ebx, $_ID_1
    jmp  END
TARGET_2:
    mov  ebx, $_ID_2
    jmp  END
    ...
END:
    nop

; if retn NN is used, remove all-1 args
    add esp, MM         ; MM = NN - 4
```

Figure 2: Translation of a return instruction.

4.1.3 Heap manipulation

Heap manipulation is a challenge when injecting a process into a set of benign processes as all access to the heap must be coordinated, simulating a single target address space among different host address spaces. If a block allocates memory using any of the standard heap functions, this memory will be valid only under the address space that blocks is executed. To overcome this problem we provide our own heap manipulation API, that will allocate shared memory regions at the same base address for all processes. This can be done by calling `MapViewOfFileEx()` with a non-NULL `lpBaseAddress`.

During the translation we check for heap management functions like `malloc()`, `calloc()`, `LocalAlloc()`, or `HeapAlloc()` and replace the call with an emulator-local alternative that is aware of the translation. Similar work is done for other heap management functions, like `LocalFree()` or `MapViewOfFile()`.

4.1.4 Socket descriptors and HANDLEs

The biggest challenge for the malWASH implementation is to transparently support HANDLEs, HKEYs (essentially a HANDLE), sockets descriptors and FILE* pointers (called "descriptors" for simplicity). Descriptors are unique per-process. If process A creates a socket, process B cannot use that socket, even if it knows the socket descriptor. However there are two functions provided by the Windows API, `DuplicateHandle()` and `WSADuplicateSocket()` that duplicate a HANDLE and a socket respectively. Unfortunately, there is no native support for duplicating FILE* pointers. We discuss support for FILE pointers in Section 4.3.2.

Descriptor support has both an offline and an online component. Our IDA Pro plugin searches for calls to descriptor functions (complete function declaration is proviced) and marks them and their parameters for further analysis.

If a block creates, duplicates, or deletes a descriptor, this information is propagated to all other emulators using the corresponding calls. The emulator includes runtime functionality to coordinate this information.

4.2 Phase 2.a: Loading emulators

The loader is the first part of malWASH that executes. It initializes the required shared memory regions (administrator privileges are required to set up shared memory, obtaining these privileges is orthogonal to malWASH) and finds up to *N* processes to inject the emulator. The standard code injection involves four functions: `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. Calling these functions in that order is suspicious.

Although we cannot avoid to call these functions in that order, we make detection harder in two ways. The first is to recursively use the chopping idea of malWASH: the loader spawns 3 new processes. Each of these processes calls exactly one of the four functions and informs the next one to continue. HANDLEs can be duplicated using DuplicateHandle() and shared with any Inter Process Communication (IPC) mechanism. This does not solve the problem but it adds one more layer of indirection. The second way we make detection more difficult is to use equivalent undocumented functions from the NT API: `ZwOpenProcess`, `ZwAllocateVirtualMemory`, `ZwWriteVirtualMemory`, and `NtCreateThreadEx`. Both `CreateThread` (a benign function) and `CreateRemoteThread` (a notorious function), internally call `NtCreateThreadEx`. Thus a detection tool has to check the function arguments to decide if a call is malicious or not, resulting in performance overhead.

If these mitigations are not enough, the loader can spawn new processes, instead of infecting existing ones, or infect non-running processes using one of the existing methods viruses use for injection [29]. These approaches are not a panacea against detection and we assume that the loader is, for now, trusted.

4.3 Phase 2.b: Executing the binary

After the loader finishes, it exits, and the emulator starts executing the individual pieces of program code, emulating a regular process environment. The emulator runs under a foreign process, like a parasite, and has no knowledge of the environment during start up. This makes the development of the emulator an extremely challenging process. Written in pure assembly, the emulator consists of 5,500 lines of assembly code (less than 14 kB of compiled code) and can execute all the blocks in the correct order.

4.3.1 Core environment

When the emulator starts executing it must first establish its execution environment. By reading the Process Environment Block (PEB), the emulator finds the entry point of kernel32.dll and the address of `LoadLibrary()` and `GetProcAddress()`, allowing us to find all other addresses in the system. The emulator then queries for a (randomly) named shared memory region that contains the emulator state and the shared heap.

To get to an executable state, constant addresses to segments must be relocated to shared regions and functions must be resolved to actual addresses, except some special functions (e.g., those in Section 4.1.4) that are redirected to internal functions of the emulator.

The emulator keeps "virtual registers" that the original binary will use. Context switching is done before and after block execution. In each iteration the emulator waits on a semaphore to get a mutual lock to execute the next block. When it takes the lock it copies the next block into a local buffer. Eventually, the emulator will start executing the block using the virtual registers. When the block finishes, the `ebx` register will contain the next BID and control returns to the emulator to dispatch the next block.

There is also a special shared region, called *Shared Control Region*. This region coordinates all emulators and contains (among other fields) the virtual registers. Stack is handled like other segments. During startup, the emulator sets the virtual `esp` and `ebp`, with the value of the shared stack, so the malware will not see any difference and will use the shared stack instead. The loader prepares any command line arguments of the original program on the stack.

4.3.2 Advanced Components

So far, the emulator can execute a program under multiple address spaces but there are many small details that may cause the execution to fail. Here we discuss and address these problems.

All emulators need to communicate. We therefore reserve some space in the shared control region and use it

duptab (duplication table)

original value	type	reserved	P1 handle	P2 handle	...	Pn handle
0x000004c8	SOCK	0	0x000004c8	0x000006c8	...	0x000008c8
0x00000504	HDL	0	0x000004bc	0x00000504	...	0x000008c0
....						
0x0000060c	SOCK	0	0x0000060c	0x00000700	...	0x000009a8

Figure 3: An instance of duptab

as a mailbox. Emulators communicate by sending messages, each message consists of a header followed by the data (a message looks like an UDP packet). Emulators check their mailboxes (e.g., they simply read the value of the mail counter) and process any messages, before execution of each block.

Section 4.1.4 discussed the challenge of duplicating descriptors between emulators. The offline part replaces the use of the descriptors with calls into the emulator. Here, we discuss the implementation of these functions. We allocate a table (called Duplication Table or "duptab") with function pointers for each of the internal descriptor functions and dispatch the functions accordingly. An instance of duptab is shown in Figure 3.

Duptab contains one row for each descriptor that the original binary uses. Each row contains the original value, the type (socket, HANDLE, or HKEY) of the descriptor, and the value of the duplicated descriptor for each host process. The emulator functions then use this table to translate a descriptor to the local descriptor.

Unfortunately, there is no mechanism to duplicate FILE* pointers. We solve this problem by using an alternative approach: We provide our API replacements for functions that use FILE* pointers. These replacements are simple wrappers of equivalent functions that use HANDLEs (which we can duplicate). E.g., `fopen()`, is a wrapper for `CreateFileW()`, `fprintf()` is a wrapper for `sprintf()` and `WriteFile()` and so on.

Beyond FILE* functions, several other functions need replacement. For instance, if the original binary calls `ExitProcess()`, we terminate all emulators (instead of terminating the current process). The emulator keeps a list of such special functions and replaces them with the internal implementations during startup. Other types of functions that need replacements are: functions that perform per-process specific actions (e.g., `SetCurrentDirectory()`) or functions that keep internal state (e.g., , `strtok()`). In both cases, the emulator has to replicate the information across all emulator instances.

There are some sequences of functions, that

must be executed in the same address space, e.g., {bind, listen, accept} and {GetStartupInfo, CreateProcess}. If listen() is executing in a different address space than bind(), even though the socket is successfully duplicated, an WSAEINVAL error will be returned (this is a Windows bug). Our emulator uses a *call cache* to address this issue. Each function in a chain is marked as *push* while the last one is marked as *sweep*. Replacements are provided for these functions to include the push-sweep functionality. An emulator does not execute a *push* function; instead it pushes the function (with its arguments) on the *call cache* and returns a fake successful value. When an emulator finds a *sweep* function it executes all functions from the *call cache* along with the last one, flushing the *call cache*. Although not perfect, this approach works well in practice.

The distributed design of malWASH allows us to handle multi-threaded programs by creating a shared stack and virtual registers for each thread. Each thread contains its own semaphore and its own variable that indicates the next block. Each emulator uses a round-robin algorithm to execute blocks from all "RUNNING" threads. Simple replacements are also provided for thread management functions: CreateThread() and ResumeThread() mark and emulated thread as "RUNNING", ExitThread() marks it as "UNUSED" and SuspendThread() marks it as "SUSPENDED".

The job of the emulator is twofold; it executes the emulated binary and keeps itself stealthy. Emulators can "ping" other emulators to see if all of them are alive. When an emulator detects that some are missing, it could invoke the loader to inject the missing emulator into a new process.

4.4 Recovering terminated instances

The core functionality of malWASH is to ensure that the original binary executes as if being run in a regular environment. In addition, malWASH also ensures resilience and recovery against "attacks".

Resilience, is enforced as a side effect of malWASH's distributed nature. We may run into the problem that an analyst kills all but one emulator instances to simplify the analysis process. Therefore, malWASH also needs a *recovery* mechanism. We already have a communication mechanism between emulators (Section 4.3.1) and as we mentioned in Section 3, the total number of running emulators is constant and known to all emulators. Thus, checking whether an emulator was killed is straight forward: each emulator periodically sends *heartbeat* messages to all emulators. If an emulator stops receiving heartbeats, it can invoke the loader process again, to respawn the missing emulators.

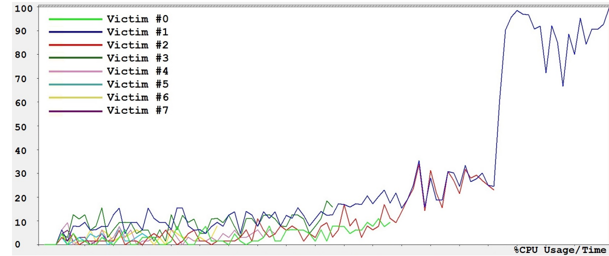


Figure 4: CPU usage among infected (idle) processes

5 Evaluation

We evaluate malWASH by targeting a set of malware samples that we inject into the most popular browsers (Google Chrome v50.0.2661.94, Mozilla Firefox 6.0.1 32 bit, Opera 12.16 and Safari 5.1.7) as victim processes under the Windows 8.1 Pro x64 operating system. Chrome's security feature of separating each tab as its own process comes in handy and allows malWASH to inject a different set of chunks into each per-tab process and shared memory regions across Chrome instances will not raise alarms.

Table 2 shows details of the malware samples we evaluate. The total number of instructions is not equal to the number of blocks in paranoid mode as malWASH omits code before and after main() as the malWASH loader component sets up the process environment and not the initialization code in the executable.

We inject malWASH into 1, 2, 4, and 8 Chrome processes, executing the samples in the different modes. In all cases both the host processes and the emulated process run without error. The host process continues without measurable performance degradation.

5.1 malWASH resilience

Due to the distributed nature and the shared state of malWASH, killing an emulated process is hard. In Figure 4 we inject a sample into 8 idle processes (so any CPU usage will come from malWASH) and start measuring their CPU usage using Microsoft Performance Monitor. Initially, all host processes execute roughly the same number of blocks, so the CPU per host process stays low. As we kill off individual host processes, the remaining emulators end up executing more blocks, increasing their CPU usage. If additional stealth is required, the emulators can throttle execution of the target process and add sleep intervals between block executions.

5.2 Case Study: Remote Keylogger

For malWASH we assume that the target process is not CPU intensive. For CPU intensive workloads, the emu-

Sample name	Type	## Instructions	Blocks Generated		
			BBS	**BAST**	**Paranoid**
Trojan.Win32.Keylogger.Gen	keylogger	2957	347	541	1484
Trojan.Win32.Invader.aa	backdoor	6359	118	233	782
Gen:Heur.Bodegun.8	backdoor	1326	112	195	496
Virus.Win32.FileInfector	virus	1739	98	183	772
TrojanSpy:Win32/Keylogger.BZ	keylogger	1380	89	178	546
Trojan-Spy.Win32.DiabloII.a	trojan-spy	162	62	86	162
W32/S-ac5b79f0!Eldorado	trojan	1837	67	141	431
W32/SelfStarterInternetTrojan!M	trojan-backdoor	3391	107	209	576

Table 2: Block statistics of malware samples.

lator may be an issue as there is overhead between executed blocks. Our emulator works well for programs that require stealthiness with little computation. Examples of such programs are keyloggers or host-based backdoors. In this section we focus on a remote keylogger to demonstrate the effectiveness of malWASH.

The remote keylogger works a follows: it opens a TCP connection to a remote host and sends captured keystrokes to the host. For the evaluation, the keystrokes were sent to a different process on the same machine. The target program is repeatedly checking whether the foreground window contains keywords from a whitelist (e.g., Facebook, GMail, Hotmail, or Twitter). And if so, it starts keylogging by checking the state of each key.

We measured performance impact by using the Octane 2.0 JavaScript benchmark on the host browsers' processes. In this benchmark we inject malWASH into the browser process that runs the benchmark for each experiment. Table 3 shows the average and standard deviation of the benchmark scores for five runs, the low standard deviation shows that the results are stable. The difference of the performance results across injected and non-injected version is in the noise and will make intrusion detection based on performance results hard.

Figure 5 shows a second scenario where we inject the keylogger under malWASH in one Firefox process and four Chrome processes (Chrome has four running processes even with a single open tab), measuring their CPU usage using the Microsoft Performance Monitor. During normal browsing we observe some spikes due to regular browsing activity. Then we stop browsing (browsers are idle) and inject malWASH. At this point there is a small peak due to malWASH startup. As browsing continues, the keylogger now runs inside the host processes and captures keystrokes. After some time we close Chrome and the emulator inside Firefox now has to execute all blocks, showing a slight increase in CPU usage for the Firefox process.

This benchmark shows that we can distribute the load of the emulator across several processes. With an increasing amount of host processes, the overhead for each individual host process through the injected process is reduced.

5.3 Discussion

Detecting programs running under malWASH through static or dynamic analysis is difficult. Static analysis is complicated because the original binary is chopped into many small pieces, likely below the signature threshold. The (tiny) emulator itself can also be protected using existing (automated) diversity techniques. Dynamic analysis is challenging as the behavior of the target program is hidden under the infected processes, making it hard to observe a sequence of calls of the target program. Therefore, defenders will likely move towards detecting malWASH instead of the target program. This by itself has the advantage of hiding the true functionality of the emulated program.

5.3.1 Protecting the emulators

Although existing detection methods will have a hard time detecting the original binary, they can be used for detecting the emulators. We argue that behavioral analysis of emulator is challenging because: (i) the emulator is very small (14kB), (ii) the emulator uses only a tiny set of system calls (for shared memory management) which will appear benign, and, most importantly, (iii) these system calls are well mixed with a subset of system calls from the emulated binary. In addition, the emulator can leverage any existing obfuscation techniques to make analysis harder.

An issue that the emulator faces is that it uses dedicated threads with similar behavior. Thus, instead of a per-process analysis, a defender could look at the actual threads and try to identify emulator threads. However, this situation is somewhat similar to the status quo: mal-

Average scores from Octance 2.0 Javascript Benchmark												
	Google Chrome			*Mozilla Firefox*			*Opera*			*Safari*		
Mode	w/o	Std	Full	w/o	Std	Full	w/o	Std	Full	w/o	Std	Full
Average	19,541	15,762	11,226	16,259	12,146	10,356	6,048	4,832	3,988	3,163	2,328	2,041
St. Dev	316	754	1,431	947	2.727	650	201	250	136	99	153	38

Table 3: Statistics from running the Octane 2.0 JavaScript benchmark five times in each of the most popular browsers, "w/o" shows execution without injection, "Std" shows a keylogger that scans for keywords for half of the time and captures and sends keystrokes for the rest of the time, while "Full" shows the keylogger capturing and sending keystrokes 100% of the time.

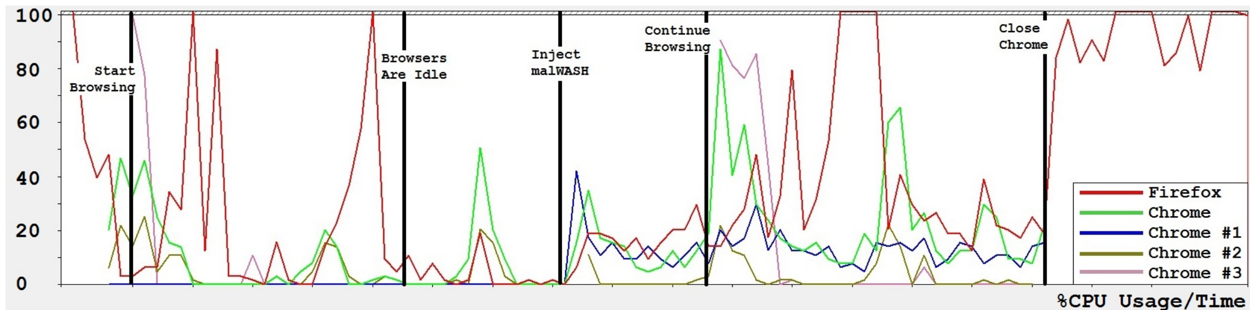


Figure 5: CPU usage of Firefox and Chrome under malWASH infection

ware uses a dedicated process within the system. One option would be to chop the emulator itself into small components, injecting them into different threads of the same process. This would lead to yet another (smaller) sub-emulator. Sub-emulators are much simpler because they run under the same address space and thus they lack the aforementioned problems that malWASH tries to solve. No shared memory is required, just a form of synchronization (e.g., spin locks or covert channels), hardening the options for behavioral analysis and spreading the emulator across several threads.

5.3.2 Fixing any abnormal system behavior

The performance overhead for malWASH is small for non-CPU intensive workloads (see Figure 4). A possible detection mechanism could spread "honeypot" processes that are idling on the system. As soon as the emulator is injected into these processes they will start to execute *some* computation and the malWASH injection can be detected. malWASH can try to mitigate by scanning for active processes by making the loader more complex (and therefore more detectable).

Careful selection of host processes, hides potential behavioral discrepancies of a process, e.g., no alarms are raised for an emulator that opens a remote connection if it is running in a browser. Process selection is an open problem and we leave it as a future work. In short, malWASH could observe the behavior of a process, and if suitable, do the injection.

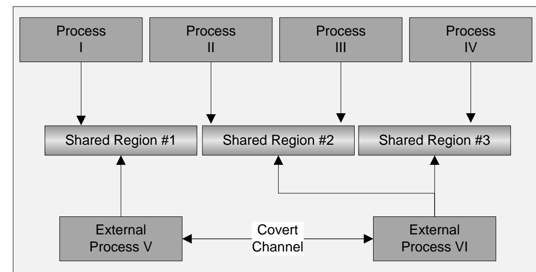


Figure 6: Thwarting detection based on shared memory correlation. Here processes I through IV used to share the same mapping. We create 3 replicas for the shared mapping with two processes attached each.

Another opportunity to detect malWASH is the shared memory regions. A detection mechanism may correlate host processes through their shared memory regions. On one hand correlation is challenging, due to the large amount of shared memory regions that are active across all processes on windows systems. In addition, malWASH does not require a star-like mapping where the same shared memory region is mapped among all processes (even for heap allocated shared regions) but can also use duplicated regions as shown in Figure 6.

With duplicated regions, we maintain multiple copies of the same shared mapping, and we force at most two

processes to share the same region. Each region could then use a disjoint encryption key to avoid correlation between shared regions. In order to keep these shared regions consistent, some "external" processes are needed. Each external process is responsible for keeping the subset of shared regions consistent. External processes communicate with each other to keep their subsets consistent. This communication is done using covert channels or by reading/writing regions to temporary files to avoid "circles" of processes connected by shared memory.

In case that usage of shared memory is a problem by itself, it can safely replaced by different (and admittedly slower) mechanisms like files, pipes, or covert channels.

Also, the distributed nature of malWASH does not require all the blocks and program's state to be present in memory during execution: the emulator could request the next block and the current program's state from a remote host which is controlled by the bot of the attacker.

As discussed in Section 4.2, the loading is the most exposed part of malWASH. If our proposed obfuscation approach is not stealthy enough, additional emulator processes can be spawned on demand, further obfuscating the loader.

We do not claim that this section covers all methods to detect malWASH and other ways may exist. The current prototype of malWASH is not complete but focuses on showcasing the technique. Overall, malWASH is a new technique to hide a target program in a set of benign processes.

6 Conclusion

Hiding processes in an execution environment is a challenging problem. While static detection is straightforward to evade using metamorphism [33] and diversity, dynamic detection can single out processes at runtime due to their behavior.

We present malWASH, a tool that hides the behavior of an arbitrary program by distributing the program's execution across many processes. We break the program into small chunks and inject these chunks into other processes. Our emulator captures and synchronizes state among the processes and coordinates the execution of the program, hopping from process to process and weaving individual instructions and system calls into the stream of instructions and system calls of the host program. We also propose the use of sub-emulators to further protect malWASH itself.

Our evaluation shows that our prototype of malWASH successfully distributes different malware programs into sets of benign processes. Detecting coordinated small chunks of malicious code in benign processes is a challenging problem for the research community.

7 Acknowledgements

The present work was supported, in part, by NSF CNS-1513783 and the "Andreas Mentzelopoulos Scholarships University of Patras".

References

[1] John Aycock, Rennie deGraaf, and Michael Jacobson Jr. Anti-disassembly using cryptographic hash functions. *Journal in Computer Virology*, 2006.

[2] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2006.

[3] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and antivm technologies.

[4] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. *NDSS*, 2014.

[5] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. *ASIACCS*, 2011.

[6] Dhruwajita Devi and Sukumar Nandi. Pe file features in detection of packed executables. *International Journal of Computer Theory and Engineering*, 2012.

[7] Stephen Dolan. mov is turing-complete. `http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf`, 2013.

[8] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press; 2 edition, 2011.

[9] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 2012.

[10] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley; 1 edition, 2005.

[11] Peter Ferrie. Attacks on virtual machine emulators. *Symantec Security Response*, 2006.

[12] Stephen Fewer. Reflective dll injection. `http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`.

[13] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. *DSN*, 2015.

[14] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[15] Xin Hu, Tzi cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. *ACM Conference on Computer and Communications Security*, 2009.

[16] Emily R. Jacobson, Andrew R. Bernat, William R. Williams, and Barton P. Miller. Detecting code reuse attacks with a model of conformant program execution. *ESSoS*, 2014.

[17] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. *USENIX Security Symposium*, 2009.

[18] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *ournal of Machine Learning Research*, 2006.

[19] Wenke Lee, Salvatore J. Stolfo, and Philip K. Chan. Learning patterns from unix process execution traces for intrusion detection. *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[20] Metasploit. `https://www.metasploit.com/`.

[21] Vishwath Mohan and Kevin W. Hamlen. Frankenstein: Stitching malware from benign binaries. *Usenix WOOT*, 2012.

[22] Vasilis Pappas. kbouncer: Efficient and transparent rop mitigation. *Usenix Security Symposium*, 2013.

[23] Michalis Polychronakis and Angelos D. Keromytis. Proceedings of the 5th international conference on information systems security. *MALWARE*, 2009.

[24] Michalis Polychronakis and Angelos D. Keromytis. Rop payload detection using speculative code execution. *MALWARE*, 2011.

[25] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. Ropinjector: Using return oriented programming for polymorphism and antivirus evasion. *Blackhat USA*, 2015.

[26] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dssel, and Pavel Laskov. Learning and classification of malware behavior. *DIMVA*, 2008.

[27] Monirul I. Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip A. Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. *ESORICS*, 2008.

[28] P. V. Shijoa and A. Salimb. Integrated static and dynamic analysis for malware detection. *ICICT*, 2014.

[29] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[30] David Wagner and Drew Dean. Intrusion detection via static analysis. *IEEE Symposium on Security and Privacy*, 2001.

[31] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. *IEEE Symposium on Security and Privacy*, 1999.

[32] Mark Vincent Yason. The art of unpacking. `https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf`, 2007.

[33] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.

[34] Syarif Yusirwan, Yudi Prayudi, and Imam Riadi. Implementation of malware analysis using static and dynamic analysis method. *International Journal of Computer Applications*, 2015.