Noname manuscript No.
(will be inserted by the editor)

“VANILLA” malware: Vanishing ANtiviruses by Interleaving Layers and Layers of Attacks

Marcus Botacin · Paulo Lício de Geus · André Grégio

Received: date / Accepted: date

Abstract Malware are persistent threats to any networked systems. Recent years increase in multi-core, distributed systems created new opportunities for malware authors to exploit such capabilities. In particular, the distributed execution of a malware in multiple cores may be used to evade currently widespread single-core-based detectors (e.g., antiviruses, or AVs) and malware analysis solutions that are unable to correlate data from multiple sources. In this paper, we propose a technique for distributing the malware functions in several distinct “vanilla” processes to show that AVs can be easily evaded. Therefore, our technique allows malware to interleave of layers of attacks to remain undetected by current AVs. Our goal is to expose a real menace and to discuss it so as to provide insights for the development of better AVs. We discuss the role of distributed and multicore-based malware in current and future threat scenarios with practical examples that we specially crafted for testing (e.g., a distributed sample synchronized via cache side channels). We (i) review multi-threaded/processed implementation issues (from kernel and userland) and present a multi-core-based monitoring solution; (ii) present strategies for code distribution, exemplified via DLL injectors, and discuss their weak and strong points; and (iii) evaluate how real security solutions perform when exposed to distributed malware. We converted real, serial malware to parallel code and showed that current AVs are not fully able to detect multi-core malware.

Keywords Malware · Multi-Core · DLL Injection · Cache side-channel

Marcus Botacin & André Grégio - Federal University of Paraná E-mail: mfbotacin,gregio@inf.ufpr.br · Paulo Lício de Geus - University of Campinas E-mail: paulo@lasca.ic.unicamp.br

1 Introduction

Malware are persistent threats to computer systems, and their strategies are becoming increasingly sophisticated. In recent years, computer systems have moved from single-core CPUs to hyperthread [17], multi-core [8], and GPU-based [15] systems. Since the change from 32 to 64-bit systems malware was an inevitable move [48], the same truth holds for IoT devices [42] and, why not, for multi-processed systems.

Despite most current security tools and techniques work in a linear, serial way, we are still not able to fully handle (linear, serial) malware attacks. Therefore, it is safe to think that we are not prepared to handle all harm posed by distributed and/or paralellized attacks. Vulnerabilities such as `Dirty CoW` [7] suggest we are not prepared to this type of threat at all. In addition, handling parallelism and distribution is a naturally hard task, increasing the chance of errors. Parallelism can be explored by attackers to distribute their malicious payloads into multiple small processes which looks individually unsuspicious for AV scanners but that are malicious when acting in cooperation. We dubbed this set of unsuspicious small processes that represent one bigger malicious program as “vanilla” pieces for the sake of this work since AVs would view the distributed payloads as ordinary pieces of code.

Linear and serial detectors operate by matching sequences of events (e.g., API calls) against known patterns so as to detect malicious behaviors. Splitting malicious actions into multiple processes allows bypassing this kind of detector as they are not able to group, resequence and reconstruct the serial-equivalent pattern. In fact, this reconstruction may even be impossible in most cases, given that the number of possible event-ordering combinations exponentially grows as the num-

ber of distributed malicious processes increase. While this strategy was mostly presented in theoretical studies [20, 13], it starts to appear in actual malware [43].

Whereas previous work's focus was on presenting the "in-system distributed" malware **concept**. we here study multiple, distinct techniques (e.g., locking, busy waiting and cache side-channels) for **implementing** distributed malware, highlighting the weak and strong points of project's decisions, and how they can be exploited by attackers and security solutions. More specifically, we show multiple ways of distributing the DLL injection procedure, a popular code injection technique in Windows environments. We developed a multi-core, branch-based framework to track processes' actions in multiple cores to evaluate distribute attacks and defensive measures in a real system. Furthermore, we adapt real Windows malware samples with the presented techniques to transform them into "vanilla". We then evaluate the execution of vanilla malware to discover whether current AV solutions are able to handle and detect this kind of threat or not. Our results show that current AV solutions are not able to detect in-system distributed samples even when they were able to detect their serial counterparts. In summary, our contributions are as follows:

- A security-oriented review on structures and programming practices for the development of multi-core and multi-threaded applications.
- The introduction of a distributed DLL-injection procedure that exemplifies our discussion on code distribution practices.
- The introduction of a branch-monitor-based framework for distributed code tracking, which exemplifies and supports our discussion on distributed malware detection techniques.
- An evaluation of actual AV solutions regarding their (in)ability to detect distributed malware code.

This paper is organized as follows: Section 2 shows the need of considering distributed samples in threat models; Section 3 presents background information on distributed and parallel code to support our presented developments, as well as related work and how they differ from our study; Section 4 introduces our assumptions and threat model; Section 5 presents the methodology we adopted to evaluated the developed codes; Section 6 discusses the multiple forms of distributing malicious code and possible security detection methods; Section 7 evaluated actual AV regarding their distributed code detection capabilities; Section 8 discusses our findings and their impact; Section 9 presents our conclusions;

2 Motivation

This paper's main goal is to study the security impact of multi-core-aware and/or distributed malware samples in current and future scenarios. In this section, we first demonstrate that it is possible to exploit multi-core systems facilities to bypass security solutions, motivating our work. Further, we evaluate the possible extent of damage that this type of threat may cause, thus claiming the need of enhancing existing solutions to broad their future threat models to cover this work's pointed scenarios.

An immediate malware creators' insight regarding multi-core-based systems is to check whether a given piece of code is running on a multi-processed system or not. As multi-core processors become standard, attackers may start to use this information to detect whether their code is running in a sandbox solution or in a real machine, as most security solutions, in a general way, are still single-core-based [6]. The SecVisor solution [50], for instance, explicits in its threat model: "we assume that the system has a single CPU".

Code 1 exemplifies the processor fingerprinting behavior via the use of a standard Windows API, as suggested by Microsoft [29]. It could be used by attackers to evade security solutions if they have only a single core.

```
1 #define MINIMUM_CORES 2
2 int main(){
3  int cores = GetMaximumProcessorCount(0);
4  if (cores<MINIMUM_CORES){
5    // printf("%d\n"<cores);
6    evade();
```

Code 1 Evasion of execution in sandboxes based on the number of presented cores.

To demonstrate its impact in actual scenarios, we submitted the aforementioned code example to multiple sandboxes. The output results are shown in Table 1.

Table 1 *Sandboxes fingerprinting:* all evaluated services were single-core machines whereas most actual computers are multi-core.

Service	Cuckoo [22]	Anubis [12]	Falcon [11]
Cores	1	1	1

We discovered that all online sandboxes were running on single-core machines, which is fair due to the use of virtualization for scalability of the analyses. Given this result, an attacker could refuse to run on systems which do not present a given minimum number of processing cores.

Current sandboxes and monitoring solutions are single-core-based mainly due to implementation efforts and scalability/costs issues. The first refers to the fact that Hardware Virtual Machines (HVM)-based systems, for instance, must be loaded on each core. Code transitions from one core to another require more complex algorithms to be handled than their single-core-based versions. The last, as in the malwr's case, happens due to scalability issues, since this kind of system is VM-powered, which is configured in a single-core profile to maximize computer resource usage. As a countermeasure for both cases, systems could fake CPU-related API responses, thus impersonating multi-core-based systems. However, the complexity of such decision is a critical point, because it would require faking all CPU-related APIs and not only those related to information gathering. A request to attach to a non-existing core, for instance, should be converted to a valid request. Otherwise, the request would result on a blue screen.

By checking the number of available cores for a given execution, malware samples could evade not only a malware analysis sandbox but also AntiVirus (AV) emulators, thus bypassing detection. Fingerprinting an AV emulator is a known technique [3] and its extension to cover the number of cores is straightforward. To measure the impact of this kind of technique in practice, we crawled and inspected the 226,290 most recent[1] samples uploaded to the malshare database [21], whose AVClass [47] family distribution is shown in Table 2.

Table 2 *Family distribution* in the malshare dataset.

Family	Prevalence	Family	Prevalence
Trojan	20%	Generic	15%
Adware	20%	Downloader	15%
PUA	20%	Other	5%

We inspected their function imports for thread/processor-related APIs that might indicate that the application is processor-aware. While the results shown in Table 3 can be considered only as lower-bounds for the API calls prevalence due to packing and/or runtime-generated API calls, their identification is enough to reveal whether current malware samples are already capable of exploiting distributed processing possibilities or not.

We discovered that 969 (0.43%) of the samples exhibited some thread and/or processor core-related function calls, thus suggesting that this kind of threat has a currently non-negligible, but still unexplored potential. As a side-effect of requiring and ensuring that analysis systems work in a multi-core-based way, attackers may complicate analysis tasks by leveraging distributed infection procedures, which are harder to track than their serial counterparts, as shown in the next sections. Therefore, this work shed light to possible future attackers' movements, thus allowing anticipating and enhancing incident response.

[1] December/2018

Table 3 *Processor awareness:* identified thread-related imported functions in real samples.

Function	Occurrences	Samples
GetProcessAffinityMask	800	0.35%
SetThreadAffinityMask	301	0.13%
SetProcessAffinityMask	205	0.09%
GetLogicalProcessorInformation	105	0.05%
SetThreadIdealProcessor	9	0.00%

3 Background and Related Work

We here present implementation details of multi-core/threaded applications, a broad view on thread support by the OS, their effects on monitoring systems by both attacker's and defender's perspectives, and how to attach processes and threads to specific physical cores, as required for many payload distribution tasks. We show implementations based on MS-Windows, the target OS for this work. We also present the related work and discuss how they differ from our study.

Processes and Threads Internals: The usual approach for system monitoring is to do so on a per-process basis, so one can monitor each action/process individually, using well established techniques. However, the usual way of splitting tasks into cores is by using threads. This way, we here present the differences of handling threads and processes.

Processes are identified by their unique `Process IDentifier` (PID) [26], whereas threads have a unique `Thread IDentifier` (TID) [40]. It allows enumerating and handling specific threads within processes. The functions to handle threads and processes are distinct [35], so one should properly use them according to the context. This implies, for instance, that when monitoring threaded code, one should call `GetThreadId` [31] instead of `GetProcessId` [30]. When tracing actions, one may want to map thread actions to their owner processes to match known behaviors. This task can be done by invoking `GetCurrentProcessID` [27] and `GetCurrentThreadID` [28] functions. This approach is valid from both kernel and userland. In the kernel, this strategy is leveraged by sandbox solutions [5].

Unfortunately, one cannot get the "parent" TID from the "child" TID, since there is no explicit relationship between threads. As a side effect, an early-launched thread can live even after its launcher thread exits. An exception for this rule is when the main thread exits, so all threads must be finished. It is important to notice that: (i) each process has a `WinMain` thread and possibly other endless workers associated by the compiler [45]; and (ii) the main thread does not necessarily include the C main function, but might be any other compiler-generated function including the program entry point [37].

An interesting implementation choice regards many OS schedulers, such as Linux's and the Windows' one, our focus in this work, is that tasks are scheduled by threads, and not by processes, thus allowing one to remove processes from the task list but still get them scheduled [10], an strategy often employed by rootkits. Whereas changing thread attributes is now limited by the Kernel Patch Protection (KPP) mechanism [5], it is still possible in some scenarios [9]. Whereas processes and threads internal structures have already been a target for malware attackers, we here show that the attack surface for distributed malware also encompasses exploiting thread code execution itself.

The general structure which keeps process data is the `Process Environment Block` (PEB) [33] and thread data is the `Thread Environment Block` (TEB) [34]. To get more information from a given PEB or TEB, such as command line arguments, debugger information and so on, one should query the proper structure. Threads are available both in userland, by using the aforementioned APIs, and in the kernel [32], which makes the approaches here presented portable from userland malware to kernel rootkits.

Core Switching and Attachment: An important step when developing multi-threaded threats and defenses is to attach one's code to a given core, thus allowing proper handling. An attacker might want to pin its threads to specific cores to bypass security solutions mechanisms running in a given core whereas defenders might want to pin its monitoring threads to specific cores to prevent core-migration. In addition, hardware-based monitoring features (e.g., performance counters) must be enabled on a per-core basis, so that a configuration thread must be loaded into each core [4]. Therefore, the OS must provide mechanisms to attach threads to specific cores, since the usual behavior is to schedule threads using a load-balancing policy, often resulting in core migration. Code 2 and 3 show, respectively, how to perform such thread-to-core attachment in userland and kernel spaces.

```
1 void attach_to_core(core_id) {
2  HANDLE p = GetCurrentProcess();
3  DWORD mask = 1<<core_id;
4  SetProcessAffinityMask(p,mask);
```

Code 2 Attaching thread execution to a given core: userland approach.

```
1 void thread_attach_to_core(core_id) {
2  KAFFINITY mask = 1<<core_id;
3  KeSetSystemAffinityThread(mask);
```

Code 3 Attaching thread execution to a given core: kernel approach.

Related work. Some AV reports have already identified multi-threaded malware. As an example, a report [23] presents a case in which the "malware is designed as multi-threaded application that divides process control, file infection and sending and receiving messages to and from the C&C server into different threads to share execution load". In addition, another report [25] presents a sample which launches a specific thread only to threat IoT devices. These cases, although close to this work, are distinct from the ones here tackled. We are interested in samples which launch distinct threads to evade serial, linear detectors, and not only just for performing independent, distinct tasks. More specifically, we are interested in the scenario presented in [46], which reports that "code can be separated into multiple threads of execution, which not only transforms the code, but can greatly complicate automatic analysis". In this sense, this work extends the `Shadow Attacks` concept [20], which distributes system calls over multiple processes to avoid detection. In our approach, we distribute code at API level using both processes and threads. Whereas `Shadow Attacks` demonstrated that the security solutions (sandboxes) from that time were not able to handle this kind of threat, we repeat their evaluation (as presented in Section 7) to verify whether current security solutions (AVs) evolved towards distributed malware detection. In addition, whereas `Shadow Attacks` pointed some local code distribution and detection mechanisms implementation possibilities, we dig into their details to show their strong and weak points, as presented in Section 6. Similarly, MalWasher [13] presents a solution to distribute a malicious application into multiple processes coordinated by a central entity. In this paper, we show how this coordination can be implemented (e.g., using locks, busy waiting and so on), thus demonstrating strong and weak points which can be exploited by attackers and security solutions.

4 Attack & Threat Model

In this work, we investigate how to bypass serial, linear detectors by splitting actions into multiple processes, threads or processor cores. Serial detectors identify malicious behaviors by matching a given sequence of actions (e.g., API calls) to known malicious patterns. For example, consider the DLL injection procedure, probably the most popular code injection technique used in the Windows environment both by benign applications as well as by malware samples, thus being detected by many security solutions. The serial implementation of the DLL injection procedure consists of invoking the functions in a given sequence, as shown in Code 4. This code's execution leads to the API call sequence shown in Figure 1, which can be used as a pattern for a DLL injection detector. The distribution of the DLL injection procedure will be used as example along this paper.

```
1 void inject(char *dll,int pid)
2  HANDLE proc = OpenProcess(pid);
3  int size = (strlen(dll)+1)*sizeof(char);
4  LPVOID alloc = VirtualAlloc(proc,size);
5  BOOL write = WriteProcessMemory(proc,
       alloc,dll,size);
6  HMODULE k32 = GetModuleHandle();
7  PTHREAD_START_ROUTINE tStart =
       GetThreadStart(k32);
8  HANDLE thread = CreateRemoteThread(proc,
       tStart,alloc);
9  WaitForSingleObject(thread, INFINITE);
```

Code 4 API call pattern exhibited during the execution of a serial DLL injection procedure.

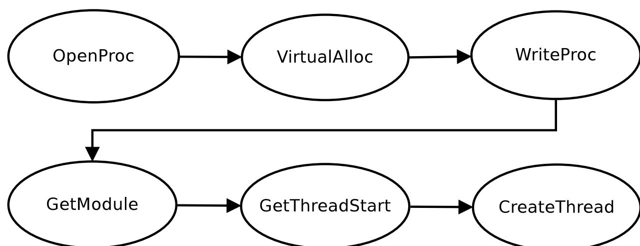

Fig. 1 Serial DLL injector. API call sequence can be used as a pattern for malicious behavior detection.

The main idea behind code splitting is that this known DLL sequence will not appear during malware execution, but multiple individual actions in distinct processes instead, which makes detection harder as grouping and re-sequencing actions in multiple processes may be unfeasible due to the exponential number of combinations, as illustrated in Figure 2.

In this paper, we investigate techniques to implement the code splitting approach. On the one hand, we show that tracking processes interactions is an effective way that security solutions can leverage to avoid handling the exponential number of combinations present when monitoring processes in the usual way. On the other hand, we show that hiding explicit processes interactions may allow detection evasion.

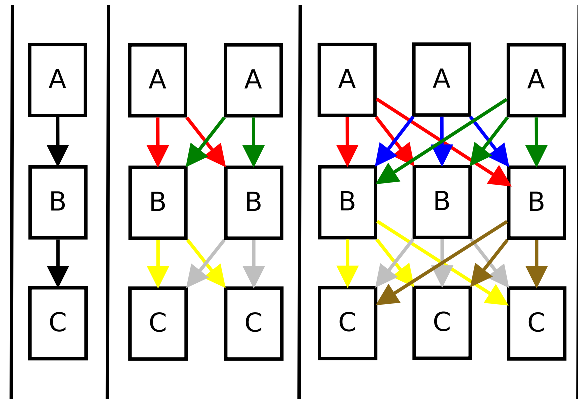

Fig. 2 Distributed pattern matching. The number of possible combinations exponentially increases when more processes are considered.

The presented code splitting techniques are intended to thwart serial, linear detectors which performs pattern matching in runtime, given the need of re-sequencing. We consider our proposed scenarios as realistic as many data-collection mechanisms operate in a per-core basis, such as approaches based in performance-counters [4]. The presented detection techniques should not be considered as the final Indicator of Compromise (IoC) for flagging an execution as a malware, but as a trigger to launch specialized scanning procedures. Finally, we do not claim that distributing malware is the only way to bypass detection mechanisms or that the hereby presented techniques are the only way to detect malware samples, but these other approaches (e.g., obfuscation) are out of this work's scope.

Our work is focused in the Windows environment, as it is the most popular [41] targeted OS by malware creators [14]. However, the approaches here presented may be applied to any OS supporting threads and multiprocessed systems. Similarly, we focus our study in userland cases, as they are prevalent. However, nothing prevents the deployment of the here presented techniques in kernel, since modern OS support kernel threads, as presented in Section 3.

5 Methodology & Tools

To evaluate the feasibility of bypassing detection mechanisms by distributing code, we first developed samples which operate in a non-serial way and further tracked

their execution using distinct techniques. All samples were developed in C, using standard system libraries, and with each thread attached to the same or to a distinct processor core, according the experiment. All evaluation and tests were performed in a 64-bit Windows 8 OS running on an Intel i7 quad-core processor.

To pinpoint the exact points which could theoretically lead to a detection, we leveraged the API Monitor tool [44] to track threads and processes at API level, thus identifying, for instance, when they enter and exit critical regions. We also leveraged the Intel PIN [18] solution to track processes and threads at instruction level, thus identifying, for instance, memory writes in foreign thread memory regions. In addition, to evaluate samples detection and evasion feasibility in a real scenario, we developed a branch-monitor-based, multi-core sandbox solution, thus simulating a security solution (e.g., AV) implemented upon a real hardware feature.

To monitor binaries which are distributed over different cores, we need to find a per-core monitoring mechanism. One of these is the Intel's Branch Trace Store (BTS), which supplies branch instructions source and target information, thus allowing binary monitoring. The use of BTS for malware monitoring was discussed in [4], but their framework implementation is single-core based. We extended their framework to work on a multi-core basis by enumerating all processor cores, launching a thread for each one, attaching them to their respective cores, and enabling the BTS mechanism in each. In addition, we modified their interrupt handling routines to operate in the fixed mode, as the original framework handles interrupts by relying on a Non-Maskable Interrupt (NMI) handler and these interrupts must not overlap, which may happen when BTS is enabled on multiple cores. Once an interrupt is handled, we isolate processes in the same way as performed in their implementation. In addition, we also recover in which core the given interrupt is happening by calling `GetCurrentProcessorNumber` [38]. From this modified framework, it is straightforward to to track processes, even during core switches, as both the executing core and PID information are provided by the framework. The algorithm to track core switches is shown in Code 5.

```
1 def callback(CURRENT_PID,CURRENT_CORE):
2  if last_core[CURRENT_PID]!=CURRENT_CORE:
3   detect_switch_from_to(last_core[
        CURRENT_PID],CURRENT_CORE)
4   last_core[CURRENT_PID]=CURRENT_CORE
```

Code 5 Monitoring process core switches.

6 Distributing Malware

In this section, we present distinct methods and strategies for pattern matching bypass and code distribution of malware. We discuss their effectiveness regarding several detection techniques, and consider the distribution of the DLL injection procedure to exemplify the implementation of all proposed distribution methods and strategies. We also show how a specially-crafted code that distributes its operation among system cores can bypass malware detection mechanisms.

The main idea of distributing the DLL injection procedure is to to split its internal functions in distinct threads and/or processes. For the sake of simplicity, we opted to implement a minimal 2-component model example on which each one of these components execute a single step and waits for the result from the other thread, as shown in Figure 3.

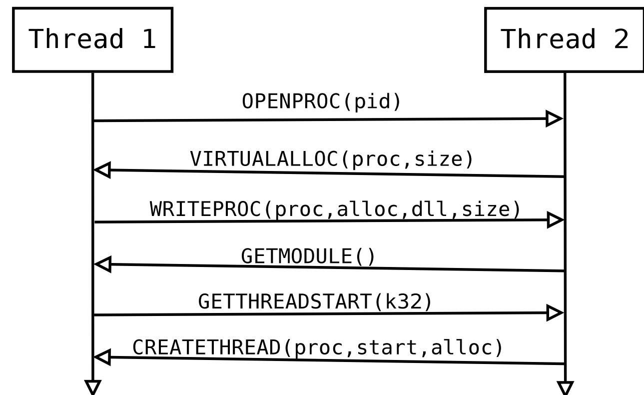

Fig. 3 Splitting DLL injection code into 2 components. Each component executes a distinct step and waits for the other one to proceed.

We highlight that the steps can be distributed and combined between the two components without presenting a clear pattern (e.g., half actions each one, a third and two third, interleaved steps etc.), which crash serial pattern detectors and force security solutions to consider all possible combinations when trying to group and re-sequence the executed functions.

6.1 Lock-based, Distributed DLL Injection

A straightforward way to synchronize the steps between two threads is to use locks (e.g., mutexes, condition variables, and so on), as here presented.

Implementation: We opted to synchronize the two threads using condition variables. We notice that, as

soon as the Thread 1 (Code 6) completes its task, it wakes up Thread 2 (Code 7) and waits for its response.

```
1 DWORD WINAPI T1(...){
2 pcontext ctx = (pcontext)lpParam;
3 EnterCriticalSection(&ctx->cs);
4 ctx->proc = OpenProcess(ctx->pid);
5 LeaveCriticalSection(&ctx->cs);
6 WakeAllConditionVariable(&ctx->cv2)
```

Code 6 Lock-based, Parallel DLL injection. Thread 1 code excerpt.

Thread 2, in turn, starts waiting for Thread 1's first step, then computes its task when woken up.

```
1 DWORD WINAPI T2(...){
2 pcontext ctx = (pcontext)lpParam;
3 SleepConditionVariableCS(&ctx->cv2,&ctx->
      cs,INFINITE);
4 EnterCriticalSection(&ctx->cs);
5 ctx->size= strlen(ctx->dll)+1;
6 ctx->alloc = VirtualAlloc(ctx->proc,ctx->
      size);
7 LeaveCriticalSection(&ctx->cs);
8 WakeAllConditionVariable(&ctx->cv1)
```

Code 7 Lock-based, Parallel DLL injection. Thread 2 code excerpt.

Detection Evasion: Figures 4 and 5 show the behavior patterns exhibited by both threads (T1 and T2) during their execution. We notice this scheme is able to evade linear, serial detectors because the original DLL injector pattern is not exhibited.

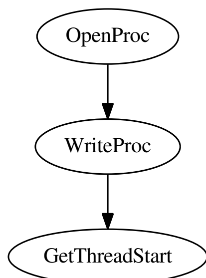

Fig. 4 T1 behavior.

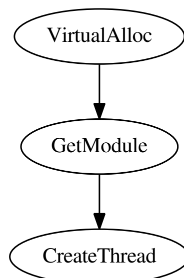

Fig. 5 T2 behavior.

Detection Alternative: Since the API calls can appear combined in exponential ways, an insightful approach to implement a detector able to reconstruct and match patterns in a distributed manner is to try not to model possible variations, but to follow the execution flow. Since a given thread must wait another to complete its task, we can observe lock patterns and transitions between them. To demonstrate this possibility, we followed critical region enters and exits using the API Monitor tool, as shown in Code 8.

```
1 1726 <time> MSVCR110D.dll
      EnterCriticalSection (0x59cbb8)
2 1727 <time> MSVCR110D.dll
      LeaveCriticalSection (0x59cbb8)
3 1729 <time> MSVCR110D.dll
      EnterCriticalSection (0x59cb90)
4 1731 <time> MSVCR110D.dll
      LeaveCriticalSection (0x59cb90)
```

Code 8 Parallel DLL injection monitoring.

We notice that the sequence of threads entering and leaving critical sections allows one to identify not only that two threads interact but also the point in which it occurs, thus enabling the correlation of the functions appearing right before one thread leaving the critical section and right after the another thread entering in the critical region. Therefore, whereas effective to bypass linear detectors, attackers may want to avoid this lock-based implementation because of this tracking possibility. In practice, however, we are not aware of sandbox solutions effectively tracking synchronization APIs, even among the mainstream ones.

6.2 Process-Based, Distributed DLL Injection

The aforementioned approach to distribute the DLL injection procedure is not limited to a thread-based implementation, but can also be implemented using processes, as here presented.

Implementation: We used the same previous step-based strategy with Inter-Process-Communication (IPC) approach. For the sake of simplicity, we opted to distribute only one task (`VirtualAlloc`), though any distribution is possible. We highlight that the absence of a single function might be enough to bypass a fixed-pattern matching detector. Notice that Process 1 (Code 9) performs all tasks but memory allocation, which is outsourced to Process 2 (Code 10).

```
1  void inject(char *dll,int pid)
2   printf("1.␣1/6.␣OpenProc\n");
3   HANDLE proc = OpenProcess(pid);
4   LPVOID alloc = VirtualAlloc_IPC(proc,
        size);
5  HANDLE VirtualAlloc_IPC(HANDLE proc,int
      size)
6   IPC data;
7   HANDLE hMapFile;
8   LPVOID map;
9   hMapFile = CreateFileMappingA(...,
        map_name);
10  map = MapViewOfFile(hMapFile,...);
11  CopyMemory(map, &data, sizeof(data));
12  data.lib=PIPC(map)->lib;
```

Code 9 IPC-based, parallel DLL injector.

To allocate memory within the open process, Process 2 has to duplicate the open handler received via IPC, thus interacting with data from Process 1.

```
void VirtualAlloc_IPC()
 hMapFile = OpenFileMappingA(...,map_name
     );
 map = MapViewOfFile(hMapFile,...);
 data = (PIPC)map;
 DuplicateHandle(hParent,data->proc,
     GetCurrentProcess(),&dup,0,TRUE,
     DUPLICATE_SAME_ACCESS);
 data->lib = VirtualAllocEx(dup, NULL,
     data->size, MEM_COMMIT,
     PAGE_READWRITE);
```

Code 10 IPC-based, parallel DLL injector.

Code 11 shows the execution output for this sample code. We notice that all steps except the second (memory allocation) are performed by the first process.

```
1. 1/6. OpenProc
1. 3/6 WriteProc
1. 4/6 GetModule
1. 5/6 GetThreadStart
1. 6/6 CreateThread
ExitCode b5570000
2. 2/6 VirtualAlloc
```

Code 11 IPC-based, parallel DLL injection output.

Detection: The IPC-based approach presents the same monitoring drawback as the threaded one, because the call for the `DuplicateHandle` function can also be tracked, thus revealing the interactions between the two components. From an attacker's perspective, the smaller analysis surface the better, thus avoiding such locks and handler duplication is desired.

6.3 Busy-waiting, Distributed DLL Injection

We previously presented how splitting functions into threads can make detection harder. The lock-based implementation, however, leaks flow information at API level. To avoid the API calls related to thread synchronization, we propose using busy waiting as synchronization mechanism. This way, the threads get synchronized by polling a global memory value which mimics the API-based lock. This approach can be understood as an spin-lock [39] implementation without relying in any system API to avoid API tracking.

Whereas benign application implementations usually prefer to wait on alertable locks and avoid busy waiting due to performance issues [36], this is a matter about malware samples do not care about. Some samples may even intentionally cause performance degradation to implement stalling behavior [16]. Therefore, implementing busy-waiting synchronization procedures seems to be a viable and straightforward strategy for malware samples.

Implementation: Replacement of the condition variables present in the lock-based approach by two boolean variables to control threads' steps. Thread 1 (Code 12) starts its execution opening a handle to the target process. It then sets its own variable as as busy (false) and externally set Thread 2's variable as ready to run (true). It then waits in a loop for Thread 2 restoring Thread 1's variable as ready-to-run. This procedure is repeated until all steps are completed.

```
DWORD WINAPI T1(...) {
  pcontext ctx = (pcontext)lpParam;
  ctx->proc = OpenProcess(ctx->pid);
  ctx->busy1=FALSE;
  ctx->busy2=TRUE;
  while(ctx->busy2==TRUE);
  ctx->busy1=TRUE;
```

Code 12 Busy waiting, parallel DLL injection (Thread 1)

Thread 2 (Code 13), in turn, starts waiting in a loop for Thread 1 granting Thread 2 execution permission. When allowed, it performs its memory allocation task, sets itself as busy and allows Thread 1 to run, repeating the procedure.

```
DWORD WINAPI T2 (...) {
    pcontext ctx = (pcontext)lpParam;
    while(ctx->busy1==TRUE);
    ctx->busy2=TRUE;
    ctx->size= (strlen(ctx->dll)+1)*
        sizeof(char);
    ctx->alloc = VirtualAlloc(ctx->proc,
        ctx->size);
    ctx->busy2=FALSE;
    ctx->busy1=TRUE;
    while(ctx->busy1==TRUE);
    ctx->busy2=TRUE;
```

Code 13 Busy waiting, parallel DLL injection (Thread 2)

Detection: The presented approach is able to bypass linear detectors and is resistant to API-based flow monitoring detection. This approach, however, can still be tracked via shared read and writes in the same memory addresses relative to the variables which mimic the lock and its API calls. To exemplify this possibility, we implemented a multi-thread memory tracer using the PIN tool and identified the following possible accesses patterns among the two threads:

- **Reading a Not-Written memory position (RNW)**, thus indicating that a statically or non-initialized variable is being read;

- **Reading a memory position Written by Itself (RWI)**, thus indicating that a previously written value is being read;
- **Reading a memory position written by a Foreign thread Memory (RFM)**, thus indicating that the current thread is using a value set by another thread.
- **Writing a memory region for the First Time (WFT)**, thus indicating variable initialization;
- **Writing its Own Memory (WOM)**, thus overwriting a memory region previously set by the current thread;
- **Writing a Foreign thread Memory position (WFM)**, thus overwriting a memory region previously set by a foreign thread.

When running the busy-waiting distributed DLL injection on our developed monitor, it displays the output presented in Code 14. We notice that as the second thread (ID 1) is created, it reads the two foreign addresses (RFM) corresponding to the two busy wait variables, already released by thread 1 (RFT). Right after the check, it overwrites one foreign value (WFM) to set the variable to the busy state. From this point on, it starts reading its own addresses (RNW and RWI) to perform the local computations.

```
1 Created Thread 1
2 TID 1 reading foreign value at addr 0
     x7f296deccff0 from TID 0
3 TID 1 reading foreign value at addr 0
     x7f296deccff8 from TID 0
4 TID 1 overwriting addr 0x7f296deccff8
     from TID 0
5 TID 1 reading its own value at addr 0
     x7ffe8c41db28
```

Code 14 Tracking memory operations from the parallel DLL injector example.

Therefore, a possible mechanism that a security solution might implement to detect distributed malware samples is to track their interactions via shared memory accesses. From attacker's perspective, a stealth malware should avoid sharing resources among its components.

6.4 Time-based, Distributed DLL Injection

The previously presented approach is strong even when considering the synchronization API monitoring, but can be detected when memory is tracked. This monitoring is rarely found on end-user machines, but is common in analysis environments. If an attacker wants to implement a synchronization-free code, resistant even to memory tracking, a stealthier implementation method is required. Therefore, we considered replacing the busy wait variables with static timers so that the threads would not have to wait on a shared variable, but could sleep and then proceed when the timeout wakes them up. Meanwhile, the other thread must have completed its task. The sample must only assure it will not wait for a time so long that the target process could be terminated before being hijacked.

Implementation: We adapted the busy-waiting injector to wait on a timer instead on a variable. After started, Thread 1 (Code 15) opens a handle to the target process and sleeps for a predefined amount of time, which should suffice for Thread 2's next step.

```
1 DWORD WINAPI T1(...){
2  ctx->proc = OpenProcess(ctx->pid);
3  Sleep(SLEEP_TIME);
```

Code 15 Time-based synchronization for our parallel DLL Injector example (Thread 1)

Similarly, Thread 2 (Code 16) sleeps while Thread 1 executes the first injection step, and wakes up to execute the second step. This procedure is repeated until the end of all steps.

```
1 DWORD WINAPI T2(...){
2  pcontext ctx = (pcontext)lpParam;
3  Sleep(SLEEP_TIME);
4  ctx->size= (strlen(ctx->dll)+1)*sizeof(
      char);
5  ctx->alloc = VirtualAlloc(ctx->proc,ctx
      ->size);
```

Code 16 Time-based synchronization for our parallel DLL Injector example (Thread 2)

Detection: This injector implementation cannot be detected through API or memory tracking, only by pattern reconstruction approaches. For instance, when tracking the injector execution using the branch-based sandbox (Section 5), we identified the behaviors exhibited for Threads 1 (Code 17) and 2 (Code 18).

```
1 [Core 1] Injector.exe called OpenProcess
2 [Core 1] Return <ommited>
3 [Core 1] Injector.exe called lib Sleep
4 [Core 1] Return <ommited>
5 [Core 1] Injector.exe called
     WriteProcessMemory
6 (GetThreadStart)
```

Code 17 T1 distributed DLL injection detection pattern.

```
1 [Core 2] Injector.exe called Sleep
2 [Core 2] Return <ommited>
3 [Core 2] Injector.exe called VirtualAlloc
4 [Core 2] Return <ommited>
5 [Core 2] Injector.exe called Sleep
```

Code 18 T2 distributed DLL injection detection pattern.

We observe it is possible to detect the attack by associating the APIs exhibited by distinct threads. However, we benefited from knowledge about the attached cores and code behaviors to implement this detector, which is hard to be implemented in actual scenarios given the exponential number of combinations. A drawback of this approach is that it may fail due to improper thread synchronization, which can be observed in analysis environments that hook and replace timing-wise functions, such as `Sleep`, which is a frequent countermeasure to analyze time-based, evasive samples [16].

6.5 Side-Channel-Based, Distributed DLL Injector

While sleep-based synchronization allows malware samples to bypass serial, linear detectors and is not tracked by either tainting or API calls monitoring, it can be defeated in sandboxes that hook timing APIs. An alternative to synchronize the components without leveraging timing APIs is to rely on side-channels, i.e., alternative channels that may leak information that can be used to notify on system state. In particular, we propose using a cache side-channel approach in a way that a thread is notified when the other thread finished its execution step. It can be implemented by measuring cache access time and identifying variations in case another thread has changed cache state. This approach does not require any explicit inter-thread nor inter-process communication.

Implementation: The implementation of the side-channel based injector consists in replacing the busy waiting variables by causing and identifying cache side-channel effects. Thread 1 (Code 19) now presents a lock variable which is private, i.e., not shared with other threads. Upon initialization, the thread executes the first injection step (opening a process handler) and causes a cache effect (`release`), and then starts measuring further cache effects.

```
1 DWORD WINAPI T1(...){
2  /* Lock variable */
3  int my_own_lock=0;
4  ctx->proc = OpenProcess(ctx->pid);
5  /* release the other thread */
6  release();
7  /* wait in my lock until the other one
       releases me */
8  while(test_cache(&my_own_lock));
9  ctx->write = WriteProcessMemory(...);
```

Code 19 Side-Channel-based, Parallel DLL injection. Thread 1 code excerpt.

Thread 2 (Code 20) also has its own private lock. Upon initialization, it keeps checking for cache effects. When Thread 1 purposely causes a cache measure variation, this is identified by Thread 2, which then proceeds executing. After performing the second injection step (`VirtualAlloc`), the Thread 2 purposely causes a cache variation (`release`) to notify Thread 1. This step is repeated until the injection procedure is finished.

```
1 DWORD WINAPI T2(...){
2  /* lock for the second thread */
3  int my_own_lock = 0;
4  /* start locked and wait for unlock */
5  while(test_cache(&my_own_lock));
6  ctx->alloc = VirtualAlloc(...);
7  release();
```

Code 20 Side-Channel-based, Parallel DLL injection. Thread 2 code excerpt.

The routine that checks for cache variations (Code 21) is responsible for measuring the access time of the lock variable, and identifying when it takes more than expected to be accessed, which indicates that the other thread finished its step. It is supported by the fact that the lock variable was previously cached and thus it will usually take few cycles to be accessed. If the variable takes many cycles to be retrieved, it means that the cache was somehow affected by the other thread, thus causing the current thread's lock to be evicted.

```
1  BOOL test_cache(int *addr) {
2   /* get number of ticks before */
3   UINT64 start = get_ticks();
4   /* try to access the variable */
5   int tmp = *addr;
6   /* ensure instruction termination */
7   _mm_lfence();
8   /* get ticks after */
9   UINT64 stop = get_ticks();
10  /* say ok if this access was fast */
11  return (stop-start) < SLOW_ENOUGH; }
```

Code 21 Side-Channel-based, Parallel DLL injection. Checking routine.

The code which notifies about step termination (Code 22), in turn, is responsible for evicting the other thread's lock variable from the cache. We accomplished that by flushing all caches lines.

```
1 VOID release() {
2  // for each position within this array
3  for(int idx=0; idx < L1_CACHE_SIZE/
       sizeof(int); idx++) {
4   /* access the index, to ensure cache is
        filled */
5   test_cache(&m[idx]);
6   /* then flush that line */
7   _mm_clflush(&m[idx]);
8  }
9 }
```

Code 22 Side-Channel-based, Parallel DLL injection. Release.

Detection: This implementation has neither inter-threads nor inter-processes communication flows, thus its detection is not trivial. While cache eviction-based side channels can be detected by some techniques, any side-channel information may be used as alternative for thread synchronization, requiring the adoption of very comprehensive threat models by security solutions.

7 Evaluation

To evaluate the feasibility of our proposed distribution strategies in actual scenarios, we implemented distributed versions of malware found in the wild and then scanned these new versions with updated versions of commonly used AV solutions. Finally, we compared the obtained detection rates from both serial and distributed malware samples.

7.1 Distributing Real Malware

Collecting source code of real malware samples is a hard task, since most threats identified in the wild are captured as binary files. To overcome this limitation, we restricted our search to github repositories [19], from which we gathered the final set of samples considered for evaluation, i.e., those that were able to compile and execute without errors, thus ensuring their maliciousness. Below, we present the distributed implementation of three distinct malware samples. For each sample, we implemented multiple parallel versions, each one leveraging one of the distinct synchronization techniques presented in Section 6.

Alina [49] is a Point-Of-Sale (POS) malware which scraps system for credit card information. It is implemented by many C++ classes, but their main functionalities can be clustered in three groups, as presented by Alina's function-dependency graph shown in Figure 6: (i) a scanner, which inspect all processes' memory; (ii) a HTTP client to exfiltrate the collected credit card data; and (iii) a rootkit to hide its steps. Since the sample naturally presents this modular organization, we distributed it to operate with three threads, only synchronizing their execution through a barrier when each thread terminates its task.

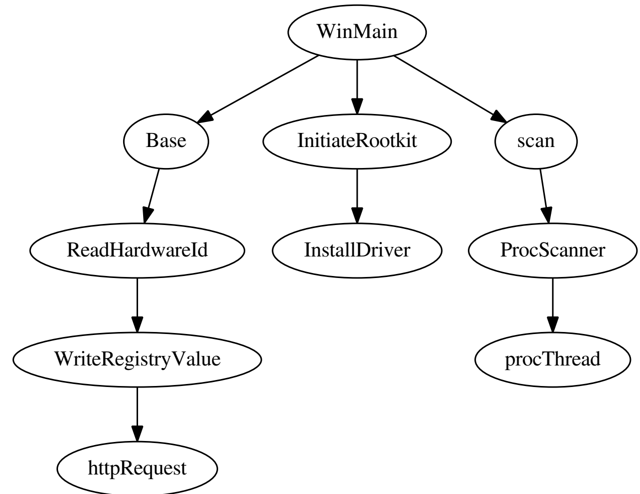

Fig. 6 Alina malware. The sample is composed of three independent components: a memory scanner, an HTTP exfiltrator and a rootkit. Our distributed version consists in implement each one of these functions in a distinct module or thread.

Dexter [51] is also a POS malware and operates similarly to Alina, scraping processes memory for credit card numbers and exfiltrating them via Internet. Contrary to Alina, Dexter already has some of its tasks distributed in independent threads. In addition, Dexter is aware of previous infections given the use of a system-wide mutex, as shown in Figure 7.

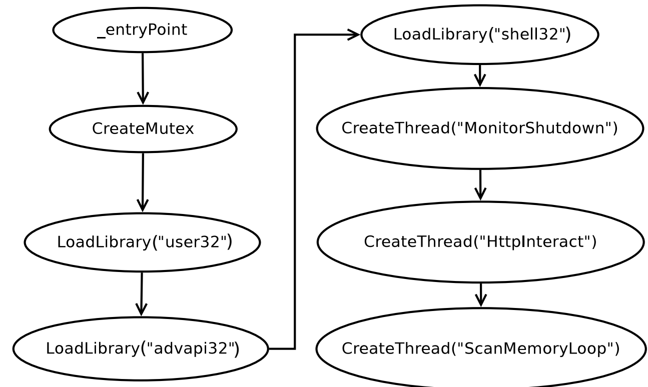

Fig. 7 Dexter malware. Serial implementation.

To distribute Dexter, as shown in Figure 8, we created mutexes for each component, so each individual run can keep track of whether its infection step was previously finished or not. We also distributed the library loading according to the requirements of each module.

As a consequence of distributing the code in multiple components, there is no explicit thread creation during malware infection steps, because the malicious procedures are inline implemented into each thread.

Trochilus [1] is a Remote Access Trojan (RAT) with multiple malicious features, from remote file upload to anti-sandbox techniques. Figure 9 shows Trochilus mal-

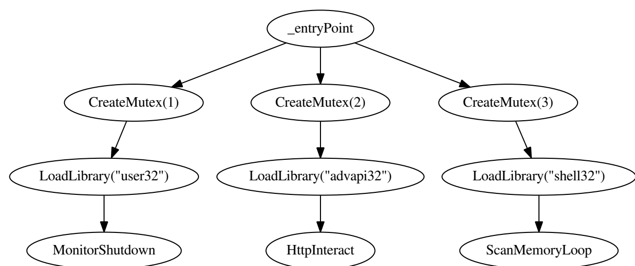

Fig. 8 Dexter malware. Each module or thread is responsible for keep track of its own infection through a private mutex, loads only the required libs for that module operation, and implements all functions in an inline way.

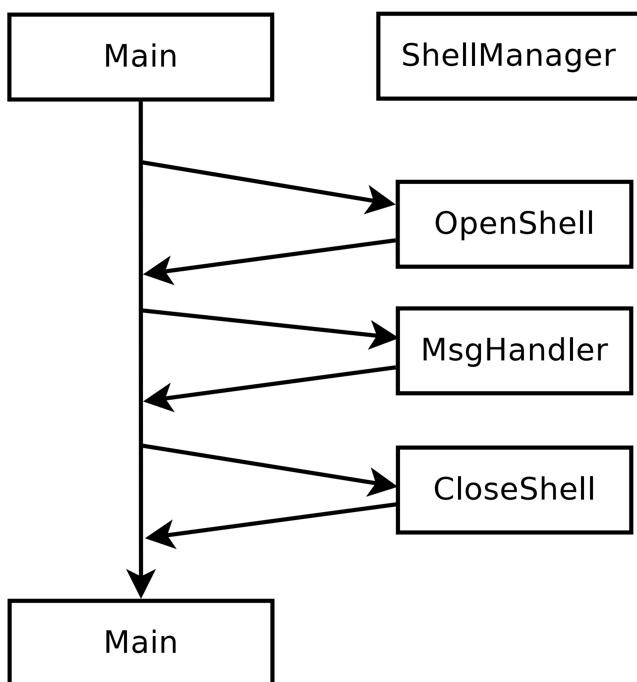

Fig. 9 Trochilus malware distribution. Interactions were split into multiple components and synchronized.

ware distribution. Since it was originally implemented as a single thread, we leverage our synchronization approaches to distribute each infection step in a new threaded or processed component.

Our experience distributing malware samples shows that, although malware constructions are diverse (e.g., synchronous vs. asynchronous constructions), all malware code could be distributed and/or parallelized. While distributing any serial code despite its internal characteristics is not often considered due to performance constraints (e.g., serial-equivalent execution), it is a feasible approach to make pattern-matching-based malware detection harder.

7.2 Evaluating Real AVs

To evaluate whether real AVs are able to handle this type of distributed malware or not, we submitted the aforementioned distributed malware samples to static and dynamic AV detection procedures. Static AV detection was evaluated first by submitting the samples to the VirusTotal service [52] and then checking their detection rates. Dynamic AV detection was evaluated by installing free versions of the five best-ranked AVs from AV-Test [2] (AhnLab, Avast, AVG, Avira, BitDefender) in a sandboxed environment, executing the malware sample with the AV turned on, and checking whether the AV detected the executed malware or not. The detection results for the serial and distributed malware versions are respectively presented in Table 4 and Table 5.

A challenge to perform a fair AV evaluation is to ensure that the targeted AV engine (e.g., pattern matching, real-time scanner, behavioral detector, etc) is triggered during an AV scan request. According our threat model, our proposed distributed malware should be able to bypass dynamic monitors, however, when an AV scan is requested, the first detection mechanism triggered is static analysis, which performs, for instance, pattern matching on the executable binary bytes and/or heuristic analyses on the binary imported libraries and functions. Table 4 shows that many AVs detected the standard (`plain`) version of the compiled code using static signature matching. Therefore, to ensure that our experiments triggered dynamic AV monitoring engines, we developed `obfuscated` versions of all malware samples by solving all symbols and strings in runtime so as to bypass any static detection procedure and force AVs to detect our samples in runtime, which effectively happened, as shown in the `Dynamic` column. As a side-effect of this experiment, we discovered that the simple fact of the same set of malicious actions being presented in distinct ways, i.e., in its serial and threaded forms, make it able to evade the detection of many static detectors. It can be observed for the Alina malware testing: Whereas 31 AVs statically detected its serial version (Table 4), only 2 AVs detected its threaded version (Table 5), despite the use of no obfuscation. Although threaded-versions of malicious code were also statically detected by few AVs, multi-processed code distributed in multiple binaries were not detected by any of them (Table 4). We hypothesize that the distribution of the code into multiple binaries makes AVs unable to detect samples by analyzing their imported libraries, which is still feasible for threaded code in the same binary.

Table 4 Serial Malware Detection. Many AVs detected the samples via static signatures, which can be bypassed via obfuscation. Alternatively, AVs are also able to detect the samples via dynamic pattern matching. Small variations in the total number of AVs are due to AV availability issues in the VirusTotal service.

Detection	**Static**		**Dynamic**	
Version	**Plain**	**Obfuscated**	**Plain**	**Obfuscated**
Malware	**Detection**			
Alina	31/57 (54%)	0/57 (0%)	5/5 (100%)	5/5 (100%)
Dexter	14/61 (23%)	0/61 (0%)	5/5 (100%)	5/5 (100%)
Trochilus	11/58 (19%)	0/67 (0%)	5/5 (100%)	5/5 (100%)

Table 5 Distributed Malware Detection. Some distributed malware samples were not detected by the same signatures which detected their serial versions. No AV was able to detect the sample using dynamic approaches, although they dynamically detected their serial counterparts.

	Static				**Dynamic**			
	Plain		**Obfuscated**		**Plain**		**Obfuscated**	
	Threads	**Processes**	**Threads**	**Processes**	**Threads**	**Processes**	**Threads**	**Processes**
Malware	**Detection**							
Alina	2/54 (4%)	0/61 (0%)	0/61 (0%)	0/61 (0%)	0/61 (0%)	0/61 (0%)	0/61 (0%)	0/61 (0%)
Dexter	3/61 (5%)	0/58 (0%)	0/58 (0%)	0/58 (0%)	0/58 (0%)	0/58 (0%)	0/58 (0%)	0/58 (0%)
Trochilus	0/57 (0%)	0/57 (0%)	0/57 (0%)	0/57 (0%)	0/57 (0%)	0/57 (0%)	0/57 (0%)	0/57 (0%)

Once a sample is not flagged as malicious by a static detection method, current AV solutions allow the application to run but keep monitoring its binary execution via runtime API invocation checks. We evaluated how AVs behave when exposed to distributed malware regarding this dynamic monitoring approach. We proceeded as follows: we (i) produced a pristine, virtualized image of MS Windows 8 64 bits and generated a snapshot; (ii) installed one of the five selected best-ranked AVs, as already mentioned above; (iii) copied the serial version of one of the three malware used for testing; (iv) made sure the AV was turned-on and double-clicked the malwre sample; (v) verified if the AV detected the running sample to finally restore the virtual machine to its pristine state. This process was repeated (steps ii to v) for each of the five AVs with each of the three malware compiled in serial, threaded, and multi-processed forms. This experiment shows that distributing malware in both `Threads` as well as in `Processes` is an effective strategy to bypass dynamic AV monitoring engines, thus demonstrating that distributed malware might pose a significant threat in a near future, therefore requiring the development of more advanced event correlation tools to mitigate the risk posed by their infections.

8 Discussion

In this work, we investigated the threat of distributed malware samples for linear, serial pattern matching-based malware detection schemes (as is the case for the widely used antiviruses, commercial or free). Although actual malware samples are not widely leveraging this kind of technique, we believe that investigating possible evasion methods is essential to prepare ourselves for attackers' next movements, as well as to proactively enhance security solutions.

Distributed malware samples were previously described in the literature in theoretical ways. Here, we advanced the discussion by presenting multiple implementation alternatives for the theoretical concepts proposed, therefore showing that in-system distributed malware is a real menace for multi-core computer systems, which are the prevalent type of device used nowadays in desktops, notebooks and mobile devices. All the development and discussion provided by our work point to the fact that project decisions taken by attackers can be used as detection triggers when this kind of threat become mainstream. Other challenges and/or observations regarding the implementation of distributed malware is discussed below.

Distributed Infection Launch. A significant challenge to deploying a distributed infection is how to launch multiple processes. A possible, but naive strategy is to pack all components within a dropper binary and extract them at runtime, which makes correlation easier as the dropped objects can be tracked through a monitoring solution. A more robust alternative would be to launch the payloads from a browser, in the form of independent processes originated by drive-by-downloads, which makes the tracking procedure harder because browsers interact with a large number of distinct objects.

Side Channels Beyond Cryptography. In this work, we presented a cache-based side-channel approach for thread synchronization. The exploitation of side-channels for security interference is popular in the cryptography context, causing keys and similar secrets leakage. Our approach extends the impact of side-channel for the malware context, thus demonstrating that its impact is broader than usually considered in most threat models. In addition, we highlight that although we demonstrated a synchronization approach based on a L1 cache, cache-based side-channel attacks can also occur at other cache levels, such as L3 [53]. In this case, since the L3 cache is shared among all cores, the attack can be leveraged from code running in distinct processor cores.
Alternative Synchronization Channels. In this work, we tackled the stealth component synchronization challenge via local system interactions. However, external entities could also be leveraged for the same task. Figure 10, for instance, shows how two threads can be synchronized via external server communications. Notice that data correlation is avoided as each thread communicates with a distinct server and they are out of band-synchronized.

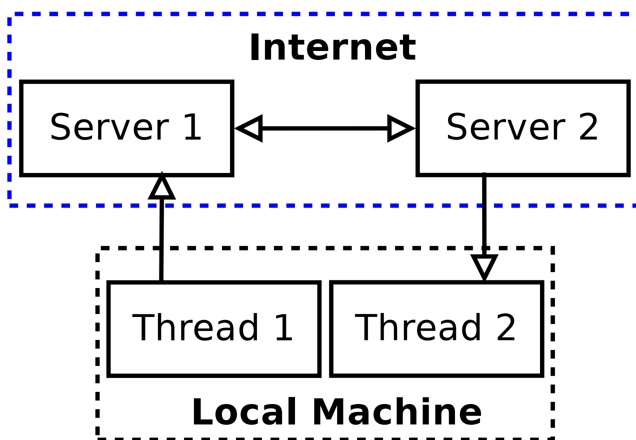

Fig. 10 External Synchronization. Data correlation is avoided as each thread communicates with a distinct server and they are out-of-band synchronized.

Limitations. Our work investigates the behavior of distributed malware samples as a future attack vector, thus our evaluation presents intrinsic limitations, since current AVs are not expected to be fully able to handle and detect this type of next-generation threat. Moreover, as AVs detection mechanisms and signatures are mostly closed source, we cannot investigate in details where they fail to detect distributed malware. Despite these facts, we believe that our work contributes to enhance incident response procedures by shedding light on the need of investigating AV detection weaknesses and pinpointing some directions on malware detection enhancements to be investigated, such as tainting and tracking data flows among multiple threads.

Future Work. We strongly believe that stressing AV solutions is essential to develop more complete defensive solutions. Therefore, we intend to develop an automated mechanism for malware distribution to allow evaluating AVs with multiple samples. In this sense, we can adapt a solution based on dependency graphs as used in parallel programming algorithms. Contrary to these, we are not limited to the parallelization of `do-all` loops [24], but we can also distribute `do-across` loops, although serially-synchronized, because our goal is not to speed up performance, but evade pattern matching procedures.

9 Conclusion

In this work, we introduced the threat of in-system distributed malware samples. We presented their weak and strong points and discussed their evolution from a theoretical perspective to a real, menacing application. We presented strategies for the implementation of distributed code at function level using distinct synchronization techniques, such as locking, busy-waiting, and leveraging cache side channels, as well as for the implementation of their detection counterparts. Finally, we evaluated the impact of this type of threat in actual scenarios by converting real, serial malware code found in the wild to parallel code, and then scanning these samples with widely used AV solutions. The results showed that current AVs are not able to fully handle distributed malware, thus reinforcing the need of enhancing distributed code handling support on security solutions.
Reproducibility. All code samples and monitoring drivers here presented are available in the github: `https://github.com/marcusbotacin/Malware.Multicore`.

Acknowledgements This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

References

1. Affairs, S.: Researchers spotted a new espionage campaign relying on a number of rats

including the powerful trochilus threat. https://securityaffairs.co/wordpress/43889/cyber-crime/new-rat-trochilus.html (2016)
2. AV-Test: The best antivirus software for windows home user. https://www.av-test.org/en/antivirus/home-windows (2018)
3. Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., Yener, B.: Avleak: Fingerprinting antivirus emulators through black-box testing. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (2016). URL https://www.usenix.org/conference/woot16/workshop-program/presentation/blackthorne
4. Botacin, M., Geus, P.L.D., Grégio, A.: Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. ACM Trans. Priv. Secur. **21**(1), 4:1–4:30 (2018). DOI 10.1145/3152162. URL http://doi.acm.org/10.1145/3152162
5. Botacin, M.F., de Geus, P.L., Grégio, A.R.A.: The other guys: automated analysis of marginalized malware. Journal of Computer Virology and Hacking Techniques **14**(1), 87–98 (2018). DOI 10.1007/s11416-017-0292-8. URL https://doi.org/10.1007/s11416-017-0292-8
6. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721, DIMVA 2016, pp. 207–227. Springer-Verlag New York, Inc., New York, NY, USA (2016). DOI 10.1007/978-3-319-40667-1_11. URL http://dx.doi.org/10.1007/978-3-319-40667-1_11
7. Dirtycow: Dirty cow (cve-2016-5195). https://dirtycow.ninja/ (2016). Access Date: 2017
8. Gepner, P., Kowalik, M.F.: Multi-core processors: New way to achieve high system performance. In: International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), pp. 9–13 (2006). DOI 10.1109/PARELEC.2006.54
9. Graziano, M.: Make dkom attacks great again. http://www.mgraziano.info/docs/graziano_hackinbo16.pdf (2016)
10. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional (2005)
11. Hybrid-Analysis: Falcon sandbox. www.hybrid-analysis.com (2018)
12. ISecLab: Anubis. anubis.iseclab.org (2016)
13. Ispoglou, K.K., Payer, M.: malwash: Washing malware to evade dynamic analysis. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (2016). URL https://www.usenix.org/conference/woot16/workshop-program/presentation/ispoglou
14. Kaspersky: Overall statistics for 2015. https://securelist.com/files/2015/12/KSB_2015_Statistics_FINAL_EN.pdf (2015). Access in May 11, 2016
15. Kindratenko, V.V., Enos, J.J., Shi, G., Showerman, M.T., Arnold, G.W., Stone, J.E., Phillips, J.C., m. Hwu, W.: Gpu clusters for high-performance computing. In: 2009 IEEE International Conference on Cluster Computing and Workshops, pp. 1–8 (2009). DOI 10.1109/CLUSTR.2009.5289128
16. Kolbitsch, C., Kirda, E., Kruegel, C.: The power of procrastination: Detection and mitigation of execution-stalling malicious code. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pp. 285–296. ACM, New York, NY, USA (2011). DOI 10.1145/2046707.2046740. URL http://doi.acm.org/10.1145/2046707.2046740
17. Koufaty, D., Marr, D.T.: Hyperthreading technology in the netburst microarchitecture. IEEE Micro **23**(2), 56–65 (2003). DOI 10.1109/MM.2003.1196115
18. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pp. 190–200. ACM, New York, NY, USA (2005). DOI 10.1145/1065010.1065034. URL http://doi.acm.org/10.1145/1065010.1065034
19. m0n0ph1: malware=1. https://github.com/m0n0ph1/malware-1 (2018)
20. Ma, W., Duan, P., Liu, S., Gu, G., Liu, J.C.: Shadow attacks: Automatically evading system-call-behavior based malware detection. J. Comput. Virol. **8**(1-2), 1–13 (2012). DOI 10.1007/s11416-011-0157-5. URL http://dx.doi.org/10.1007/s11416-011-0157-5
21. malshare: malware database. http://malshare.com/ (2018)
22. malwr.com: Cuckoo-powered malware analysis sandbox. malwr.com (2016)
23. Marschalek, M.: Analysis report. https://www.ikarussecurity.com/fileadmin/user_upload/Download/Report_MarionMarschalek.pdf (2013)
24. Mattos, L.F., Divino, C., Salamanca, J., Carvalho, J.P., Pereira, M.M., Araujo, G.: Doacross parallelization based on component annotation and loop-carried probability. In: Proceedings of the 2018 SBAC-PAD, SBAC-PAD '18 (2018)
25. Mcafee: Quarterly report. https://www.mcafee.com/br/resources/reports/rp-quarterly-threats-mar-2017.pdf (2017)
26. Microsoft: Finding the process id. https://msdn.microsoft.com/en-us/library/windows/hardware/ff545415(v=vs.85).aspx
27. Microsoft: Getcurrentprocessid function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683180(v=vs.85).aspx
28. Microsoft: Getcurrentthreadid function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683183(v=vs.85).aspx
29. Microsoft: Getlogicalprocessorinformation function. https://msdn.microsoft.com/en-us/library/ms683194(v=VS.85).aspx
30. Microsoft: Getprocessid function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683215(v=vs.85).aspx
31. Microsoft: Getthreadid function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms683233(v=vs.85).aspx
32. Microsoft: Introduction to thread objects. https://msdn.microsoft.com/en-us/library/windows/hardware/ff548146(v=vs.85).aspx
33. Microsoft: Peb structure. https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx
34. Microsoft: Teb structure. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms686708(v=vs.85).aspx
35. Microsoft: What's new in processes and threads. https://msdn.microsoft.com/en-us/library/windows/desktop/dd405527(v=vs.85).aspx
36. Microsoft: Locks, deadlocks, and synchronization. http://download.microsoft.com/download/e/b/a/

eba1050f-a31d-436b-9281-92cdfeae4b45/locks.doc (2006)
37. Microsoft: Winmain is just the conventional name for the win32 process entry point. https://devblogs.microsoft.com/oldnewthing/20110525-00/?p=10573 (2011)
38. Microsoft: Getcurrentprocessornumber function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms683181(v=vs.85).aspx (2016). Access Date: 2017
39. Microsoft: Introduction to spin locks. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-spin-locks (2018)
40. microsoft: Thread handles and identifiers. https://docs.microsoft.com/en-us/windows/desktop/procthread/thread-handles-and-identifiers (2018)
41. Netmarketshare: Operating system market share. https://www.netmarketshare.com/operating-system-market-share.aspx (2018)
42. Pa, Y.M.P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., Rossow, C.: Iotpot: Analysing the rise of iot compromises. In: Proceedings of the 9th USENIX Conference on Offensive Technologies, WOOT'15, pp. 9–9. USENIX Association, Berkeley, CA, USA (2015). URL http://dl.acm.org/citation.cfm?id=2831211.2831220
43. Prince, B.: Script fragmentation attack could allow hackers to dodge anti-virus detection. http://www.eweek.com/security/script-fragmentation-attack-could-allow-hackers-to-dodge-anti-virus-detection (2018)
44. rohitab.com: Api monitor. http://www.rohitab.com/apimonitor
45. Russinovich, M., Solomon, D.A.: Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition, 5th edn. Microsoft Press, Redmond, WA, USA (2009)
46. Sanford, M.: Computer viruses and malware by john aycock. SIGACT News **41**(1), 44–47 (2010). DOI 10.1145/1753171.1753184. URL http://doi.acm.org/10.1145/1753171.1753184
47. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: Avclass: A tool for massive malware labeling. In: F. Monrose, M. Dacier, G. Blanc, J. Garcia-Alfaro (eds.) Research in Attacks, Intrusions, and Defenses, pp. 230–253. Springer International Publishing, Cham (2016)
48. SecureList: The inevitable move - 64-bit zeus enhanced with tor. https://securelist.com/the-inevitable-move-64-bit-zeus-enhanced-with-tor/58184/ (2013)
49. Security, P.: Alina, the latest pos malware. https://www.pandasecurity.com/mediacenter/pandalabs/alina-pos-malware/ (2017)
50. Seshadri, A., Luk, M., Qu, N., Perrig, A.: Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pp. 335–350. ACM, New York, NY, USA (2007). DOI 10.1145/1294261.1294294. URL http://doi.acm.org/10.1145/1294261.1294294
51. TrustWave: The dexter malware: Getting your hands dirty. https://www.trustwave.com/Resources/SpiderLabs-Blog/The-Dexter-Malware--Getting-Your-Hands-Dirty/ (2012)
52. VirusTotal: Virustotal. https://www.virustotal.com (2018)
53. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 719–732. USENIX Association, San Diego, CA (2014). URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom