D-TIME: Distributed Threadless Independent Malware Execution for Runtime Obfuscation



Jithin Pavithran
Indian Institute of Technology Madras
jithinpavithran96@gmail.com

Milan Patnaik
Indian Institute of Technology Madras
milanpatn@gmail.com

Chester Rebeiro
Indian Institute of Technology Madras
chester@cse.iitm.ac.in

Abstract

An important aspect of malware design is to be able to evade detection. This is increasingly difficult to achieve with powerful runtime detection techniques based on behavioural and heuristic analysis. In this paper, we propose D-TIME, a new distributed threadless independent malware execution framework to evade runtime detection.

D-TIME splits a malware executable into small chunks of instructions and executes one chunk at a time in the context of an infected thread. It uses a Microsoft Windows feature called Asynchronous Procedure Call (APC) to facilitate chunk invocation; shared memory to coordinate between chunk executions; and a novel Semaphore based Covert Broadcasting Channel (SCBC) for communication between various chunk executions. The small size of the chunks along with the asynchronous nature of the execution makes runtime detection difficult, while the coordinated execution of the chunks leads to the intended malign action. D-TIME is designed to be self-regenerating ensuring high resilience of the system.

We evaluate D-TIME on a Microsoft Windows system with six different malware and demonstrate its undetectability with 10 different anti-virus software. We also study the CPU usage and its influence on Performance Counters.

1 Introduction

Fighting malware is crucial to secure computer systems. The battle can only be won if one starts to think from the other side. An important characteristic of malware is to evade detection. Over the years a variety of techniques have been developed by malware authors to hide their software from anti-malware scanners.

Primitive evasion techniques, for instance, encrypted the malign code [40] [42] [47], used a single instruction [30] [31], or adopted customized hypervisors [1]. Though these evasion techniques could hide the static content of the malware, detecting the use of such obfuscation was trivial [27] [29] [41] [43]. Modern techniques evolved to use polymorphic and metamorphic malware [50] [52] which generate semantically equivalent versions of themselves. Static

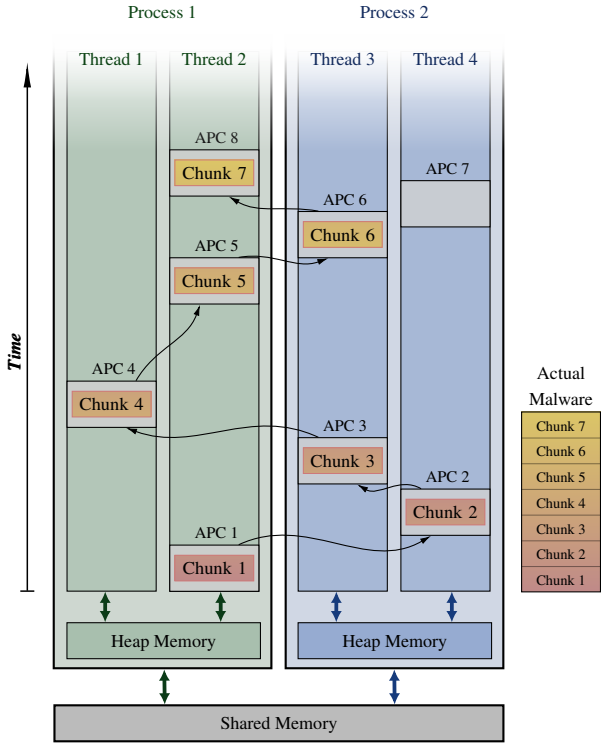

Figure 1: Illustration of D-TIME execution. Emulators get spawned as APCs in different threads under multiple processes. Chunk are executed in the emulator one at a time. Even though APCs are asynchronously triggered, robust inter-emulator communications using shared and heap memory help coordinate chunk execution to achieve the desired malign intent.

detection techniques are found to be ineffective for these malware programs compared to behavioural and heuristic techniques [23] [25], which look at runtime aspects like the sequence of API calls made [3] [35] [51], characteristics of instructions executed [44] [45] [46] and the control flow structure [4] [24] [26].

To counter behavioural and heuristic detection, attackers came up with a variety of approaches. A number of process injection techniques like DLL-injection and PE-injection

were deployed to inject the malign thread into legitimate processes [2]. These techniques camouflage the malware as a thread in the victim process. Inserting unused system calls and sleep statements were other simple methods used to evade behavioural and heuristic detection. Anti-virus software detected these techniques by isolating the program execution and filtering out the fictitious statements.

A recent powerful evasion technique distributes malware execution across multiple processes in the system. This hides the detectable behavioural patterns of the malware, making them almost undetectable. For example, in MalWASH [33], Ispoglou and Payer distribute the sequence of API calls made by the malware across multiple processes. This renders dynamic detection techniques based on aspects like the sequence of API calls and control flow unsuccessful. However, MalWASH has two major drawbacks. First, MalWASH creates multiple threads under benign processes and uses them as platforms for the distributed execution of malware. While this makes the malicious binary undetectable, the execution platforms themselves are exposed [32]. They have a common definitive signature that can be easily detected. Second, MalWASH requires administrative privileges to execute, thus reducing the scope of its use substantially.

In this paper we introduce D-TIME, a Distributed Threadless Independent Malware Execution framework, which like MalWASH provides a distributed environment for malware execution. However, unlike MalWASH, D-TIME distributes the malware execution without creating any new threads. Instead, pre-existing benign threads are used. The execution is in small bursts at a time, which are asynchronously triggered from random benign threads. This makes D-TIME considerably more stealthy. Further, D-TIME executes with normal user privileges and therefore has a wider usability.

D-TIME takes a malware executable as input and splits it into small instruction blocks called *chunks*. In the target system, these chunks are injected into threads of infected processes using a Microsoft Windows feature called APC. An APC or Asynchronous Procedural Call is a function that gets executed asynchronously in the context of a target thread [17]. Several APCs are injected into multiple target threads. Each APC is used to execute a hypervisor called `emulator`, which in turn executes a chunk from the malware executable. Emulators coordinate among themselves, regenerate and execute chunks sequentially so that collectively, the entire malicious action is performed. In order to achieve coordination, D-TIME uses communication channels and a novel covert channel called *Semaphore based Covert Broadcasting Channel* (SCBC).

For example Figure 1 shows two processes which are infected in the target system. Each process has two infected threads which trigger the emulator executions as APCs at random instances in the infected thread's context. Each emulator spawned, executes one chunk of the malware, while the communication channels ensures that collectively, emulators execute the chunks in the required sequence.

The use of APCs in D-TIME has significant advantages over MalWASH. While MalWASH needs dedicated threads to serve as execution platforms, D-TIME executes chunks of the malware in the context of the infected threads. This makes D-TIME considerably more stealthy. Further, the small chunk size and the asynchronous nature of the APC execution increases the difficulty of detection. In addition, since APCs can be invoked by user processes, D-TIME, unlike MalWASH can run without elevated privileges.

Following are the major contributions of this paper:

- We propose the first fully distributed, decentralised, threadless malware execution framework using APCs. The framework is generalisable to a large class of malware and can effectively evade detection.
- We introduce SCBC, a new robust covert channel using semaphores. Unlike most covert channels, SCBC is a persistent and broadcasting channel with high integrity of communication.
- We test D-TIME using 6 different classes of malware and evaluate the detectability with 10 state-of-the-art anti-virus software. We also evaluate the CPU usage and its effect on Performance Counters.
- We suggest countermeasures that can be incorporated in Operating Systems to detect and prevent malware that use D-TIME.

The paper is organized as follows. Section 2 discusses the necessary pre-requisites required to understand D-TIME. A brief survey of the related works is presented in Section 3. Section 4 provides an overview of D-TIME while the implementation details are discussed in Section 5. Section 6 shows the results of our evaluation of D-TIME. We discuss the countermeasures in Section 7 and we conclude the paper in Section 8.

2 Background

In this section we provide the necessary background on Windows Asynchronous Procedure Calls (APCs). D-TIME depends considerably on APCs for its functioning.

2.1 Asynchronous Procedural Call (APC)

An Asynchronous Procedure Call (APC) [17] is a function that executes asynchronously in the context of a particular thread. Every thread in Windows maintains a queue called APC-queue. Any thread in the system can queue a function in the APC-queue of a target thread where the target thread may be of another process. The queued function will be asynchronously executed in the context of the target thread.

APCs can be broadly classified as kernel-mode APCs and user-mode APCs. Kernel-mode APCs are generated by the system and gets executed by means of a software interrupt. User-mode APCs are generated by applications. They do not interrupt the thread but instead are queued in the APC-queue of the target thread. They will be unqueued and executed when the thread goes to an *Alertable Wait State* [5]. In other words, the user-mode APC is a polite way to interrupt a thread. Once a thread enters an *Alertable Wait State*, it will execute all the queued APCs in a FIFO manner. An APC can be queued using the Windows API:

```
DWORD QueueUserAPC(
  PAPCFUNC  pfnAPC,
  HANDLE    hThread,
  ULONG_PTR dwData
);
```

where `pfnAPC` defines the pointer to the function to be called as APC, `hThread` is the handle of the target thread, and `dwData` is the single parameter that is passed to the APC function. In this paper, we restrict our discussions to User-mode APCs. The techniques discussed can be directly adopted for Kernel-mode APCs but would need superuser privileges.

3 Related Work

Runtime behavioural and heuristic techniques for detecting malware have been researched extensively. However, very few published works propose means to evade these techniques. One of the most common technique to evade runtime detection is by process injection, where the malign code is injected into legitimate processes. In [2], the authors describe 10 process injection techniques for Microsoft Windows. Of these, injection using APCs is one of the most powerful. Here, a payload is queued as an APC function in the target thread. The payload when executed creates a new thread under the process that executes the malware code.

Feature distributed malware [39] was the first significant attempt to distribute malware execution. The authors, Min and Varadharajan, split the malware into sub-modules based on their functionality. These modules were then injected into appropriate processes. For example, the sub-module that leaks data can be injected into a web-browser. Later, Ispoglou and Payer, propose MalWASH [33] and showed that a fine-grained distribution can evade behavioural detection effectively. They split the malware binary into chunks of instructions and distributed the execution of these chunks in multiple processes. The objective in MalWASH was to distribute detectable patterns in malware, such as the sequence of API calls, across processes. To construct a distributed execution environment, MalWASH creates threads in multiple benign processes and executes a piece of payload called `emulator` in them. These emulator-threads coordinate among themselves to execute the chunks in the required order causing the malicious action. However, MalWASH requires administrative privileges to execute and the creation of threads required by MalWASH is easy to get detected by anti-virus software. Similar to MalWASH, D-TIME distributes and executes chunks of malware code across processes. Unlike MalWASH, D-TIME uses APCs to trigger chunk execution. Thus, D-TIME does not create threads but instead executes in the context of benign threads. This makes D-TIME much more difficult to detect compared to MalWASH. Further, D-TIME executes in user mode therefore has much wider usability compared to MalWASH.

4 The D-TIME Architecture

D-TIME has two phases. An *Offline phase*, where the malware code is split into chunks and an *Online phase*, where the chunks are executed in the target system using emulators. The emulators are injected into infected threads in the target system as APCs and are executed asynchronously when the thread goes to an Alertable Wait State (refer Section 2.1). The first emulator to execute in the system creates a communication channel, which is used for coordinating among emulators. Each subsequent emulator that executes **(1)** uses the communication channel to determine the next chunk to be executed. **(2)** It retrieves the required chunk from a secondary storage or from a shared memory and executes it. **(3)** It then updates that the chunk has been executed to the other emulators using the communication channel. **(4)** It finally spawns more emulators using the `QueueUserAPC` API (refer Section 2.1) in other infected threads in the system. The net effect is that the malware gets executed in chunks, one chunk after the other triggered randomly, in the context of different benign threads. This section provides the details of the D-TIME architecture.

4.1 Offline Processing

The primary objective of this phase is to craft chunks from the malware binary. Each of these chunks is executed by an emulator during the online phase. As an example, consider Figure 2a which shows a code snippet with a conditional statement. Depending on the truth value of the condition in Line 2 of the code snippet, either Line 4 or Line 6 will be executed. The offline phase in D-TIME splits this binary code into three chunks: `Chunk 1`, `Chunk 2`, and `Chunk 3` as shown in Figure 2b. Each chunk comprises of some malware instructions along with a few additional management instructions that are inserted to facilitate invocation of the subsequent chunks. The additional instructions in chunks (shown in red) determines the next chunk to be executed. This is stored in the `ebx` register which is used by the emulator to identify the next chunk to execute. For example, at the end of `Chunk 1`, if the value in the `ebx` register is 2, then `Chunk 2` should be executed, while a value of 3 would require `Chunk 3` to be executed next.

```
1     ...                          ;
2     cmp     [a], [b]             ; if(a==b){
3     jne     jmp  a_unequal_b     ;
4     mov     [a], 0               ;     a = 0
5  jmp a_unequal_b:                ; }
6     mov     [c], [d]             ; c = d
7     ...                          ;
```

(a) Code snippet from original malware

```
1  ------------ Chunk 1 -------------------------
2     ...
3     cmp     [a], [b]
4     jne     jmp  a_unequal_b
5     mov     ebx, 2
6     jmp     END
7  jmp a_unequal_b:
8     mov     ebx, 3
9     jmp     END
10 END:
11    nop                          ; end of chunk 1

   ------------ Chunk 2 -------------------------
1     mov     [a], 0
2     mov     ebx, 3
3  END:
4     nop                          ; end of chunk 2

   ------------ Chunk 3 -------------------------
1     mov     [c], [d]
2     ...
```

(b) Chunks after splitting

Figure 2: Example of malware code snippet getting split into chunks. A snippet containing a conditional clause is split into 3 chunks. The additional instructions added in chunk 1 and 2 (shown in red) store the identifier of the next chunk in `ebx`. The emulator uses this information to ensure the required order of chunk execution

To create chunks from the malware code, we used the techniques from [33] which proposed 3 approaches: Basic Block splitting (BBS), Below Antivirus Signature Threshold (BAST), and Paranoid mode. BBS splits the binary at the basic block level. In the Paranoid mode, each instruction of the original malware is placed in a separate chunk. While the Paranoid mode achieves the finest splitting possible, the BBS mode has the largest chunks. The BAST mode provides a flexible chunk size based on a configurable parameter. The chunk size can vary from a single instruction (like the Paranoid mode) to an entire basic block (like the BBS mode).

The size of the chunks affects the stealthiness as well as the performance of D-TIME. Small chunks distribute the malware better reducing the number of malicious instructions executed per emulator at the cost of performance overhead. Small chunks would require a larger number of emulators to be invoked to complete the malware execution thus resulting in higher performance impact.

4.2 Online Processing

The online phase is done in a compromised system. The exploit used for the compromise varies with the target and is beyond the scope of this paper. Alike [33] [34] [39] [40], D-TIME is only concerned with obfuscating the malware to prevent runtime detection.

The online phase comprises of three main components: (1) the emulators, (2) communication channels across the emulators, and (3) a Semaphore based Covert Broadcasting Channels (SCBC). Each emulator is comprised of a virtualisation layer which fetches and executes one chunk of the malware and posts the execution status over the communication channel. It also queues more emulator APCs in other infected threads. The next emulator to be triggered, will fetch and execute the subsequent chunk of the malware so that over a period of time, all the malware chunks are executed in the prescribed order.

In order to facilitate the distributed execution of the malware chunks, a communication channel is used. The channel is typically implemented using shared memory and comprises of synchronization primitives and the execution context of the malware. The synchronization primitive ensures that no two emulators simultaneously execute the same chunk. The context of the malware includes the state of registers, stack, heap, as well as an identifier for the next chunk to be executed. Thus when each emulator is triggered, it can identify the chunk to be executed, restore the state and execute the chunk.

The first emulator in the entire system creates the shared memory with a pre-defined name, `DTIME_COMM_CHANNEL`. This name should be changed randomly every time D-TIME executes on a system in order to prevent detection based on the channel name. When an emulator executes in a new infected process, it uses this name to connect to the shared memory. In each process, the shared memory gets mapped to a different virtual address range. Upon attaching, the first emulator receives the address range of the shared memory for the process. Since the emulator is a function, on function return the information about this address range is lost. This causes a problem since subsequent emulators executing in the same process will be unable to determine the address range of the shared memory. Thus, we need a way to efficiently transfer the shared memory address to all future emulators in that process.

To solve this problem, we propose a persistent covert channel using Semaphores for broadcasting the shared memory address to all the emulators that execute in the process. We call this the *Semaphore based Covert Broadcasting Channel* (SCBC). At a high level, the first emulator to execute will create a semaphore and set the count of the semaphore to a value defined by the address of the shared memory. Any other

emulator can then read the semaphore count to identify the shared memory address. The details of the implementation of SCBC is discussed in the next section.

5 Implementation aspects of D-TIME

D-TIME provides an environment for a distributed, decentralized system, where emulators are triggered asynchronously to execute malware chunks in different threads under multiple processes. There are several complications that arise from such an asynchronous emulator execution and the requirement of achieving a specific order of chunk executions. In this section, we will go over each component of D-TIME and their implementation challenges.

5.1 D-TIME Communication Channels

The communication channel for D-TIME needs to be robust so that it coordinates the distributed emulator executions and at the same time be stealthy so that it is not detected by anti-virus programs. D-TIME uses shared memory as a communication channel to implement specific chunk ordering. In order to reduce the number of accesses to the shared memory, there by reducing the chances of getting detected, we distinguish communication between the emulators within the process and emulators in different processes.

For communication between processes, we use a shared memory based on Inter Process Communication (IPC) [12] primitives. We call this as the *Primary Channel*. For communication between the emulators within a process, we use memory in the heap which we call the *Secondary Channel*. The shared and heap memories are common in various applications and therefore their use in D-TIME cannot be easily detected. The emulators use a pre-defined name such as `DTIME_COMM_CHANNEL` to connect to the Primary channel using Windows APIs like `OpenFileMapping` [13]. Once connected, the shared memory region is mapped in the virtual memory address space of the process. The secondary channels are local to each process and therefore are efficiently realised in the heap.

D-TIME can have a single primary channel or multiple primary channels. In the latter case, a subset of the infected processes share a primary channel. For example, in Figure 3, *Processes 1* and *2* share Primary Channel 1, while *Processes 2* and *3* share Primary Channel 2. Communication between *Process 1* and *3* is facilitated by *Process 2* by synchronising the contents of both shared memories. This may seem like an overhead in D-TIME, but the advantage we achieve is that it leads to more resilient design. Any channel if removed by a defender, can be rebuilt using the other existing channels. Such resiliency will not be obtained if a single primary channel is used. In this case, if this channel is removed by the defender, the entire D-TIME infrastructure will collapse.

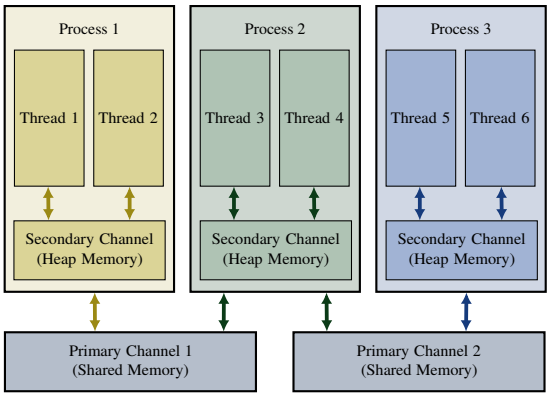

Figure 3: Design of communication channels for inter-emulator communication in D-TIME. Emulators spawn in threads across processes communicate using primary channels which are implemented using shared memory where as emulators spawn under the threads within the same process use secondary channels which are implemented using heap memory.

Without loss of generality, we discuss a single primary channel architecture in the rest of the paper, to keep the discussion simple.

Primary Channel (Shared Memory IPC): The shared memory is accessed by all emulators. It holds the following two tables: (1) *Malware Thread Table* (MTT) and (2) *Infected Thread Table* (ITT). MTT is needed for multi-threaded malware. Each row in this table corresponds to one thread of the malware and contains the identifier of the chunk to be executed next, a *context structure* containing the value of registers after the execution of the chunk for the thread, and a mutex for serialising chunk executions. The mutex should be locked by the emulator before executing a chunk and unlocked after the execution. Thus, at any given instant, only one emulator will be allowed to execute chunks corresponding to a thread of the malware.

The ITT contains the number of queued emulators waiting to be executed under each infected thread. This count is maintained to control the number of emulators queued per thread and thus the frequency of emulator execution. An uncontrolled queuing of emulators to threads will flood the system with emulators and will lead to overconsumption of CPU resources and therefore can be easily detected. Reducing the number of queued emulators would slow down the execution of malware but makes it much more stealthy.

Secondary Channel (Heap Memory): The emulator to execute first in each process will allocate memory in the heap for use as a secondary channel. This memory is used to broadcast data to emulators spawned within the process.

As discussed in Section 4.2, the problem that arises is that the emulators need to identify the address of the heap as well as the shared memory. A naïve way to solve this problem is to ensure that the heap and shared memory are always at the

same location in every process. However, this is not scalable. In the next section, we discuss an elegant solution for this problem using the Semaphore based Covert Broadcasting Channel.

5.2 Semaphore based Covert Broadcasting Channel (SCBC)

Semaphore based Covert Broadcasting Channel is based on the simple idea of using the semaphore counter as a shared integer variable. The sender creates a semaphore and sets the semaphore counter to the integer data that needs to be transmitted. The receivers can now retrieve this counter to complete the communication.

SCBC serve as a persistent channel for D-TIME. A Windows semaphore is only destroyed [38] when all of its clients detach it by calling `CloseHandle` [6] API. The emulator that creates SCBC will not detach the semaphore. Even though the emulator exits after a brief execution, the parent process is still alive. Thus the semaphore lives on. The semaphore is later retrieved by other emulators using the `OpenSemaphore` [14] API by providing the pre-defined name of the semaphore.

SCBC reduces a semaphore to a simple shared integer variable and has the following advantages over other known covert channels: *High integrity*: The integer data transmitted is always received correctly. *Broadcasting*: There could be multiple listeners for the channel. *Persistent*: The sender and receivers need not be alive at the same time. *Convenience*: It is easy to implement and as convenient as using a shared variable. The following section discusses the implementation of SCBC in Windows.

SCBC Implementation in Windows: Microsoft Windows provides APIs to create [7], acquire [19] [20] and release [18] semaphores. However, a direct API is not available for retrieving the value of the semaphore. We, therefore, read the semaphore value indirectly using the `ReleaseSemaphore` [18] API, which provides the previous value of the semaphore after incrementing it. Thus, we first call the wait API (`WaitForSingleObject` [20] or `WaitForMultipleObjects` [19] [1]) to decrement the counter by acquiring the semaphore, then we call `ReleaseSemaphore` API to obtain the previous count. One added to the previous count is the broadcasted number. The wait API call prior to the release API call balances the counter increment done by the latter. But this alone is not sufficient when we have multiple listeners. More than one listener could call the wait function before any of them call the release function. This will cause inconsistency in the counter value. However, the issue is easily solved by associating a mutex with the semaphore. Every listener is required to first lock the mutex to read the semaphore count. The mutex will be released once the value is retrieved.

[1] We may use multiple semaphores to broadcast multiple integers simultaneously. `WaitForMultipleObjects` can acquire all the semaphores together.

Algorithm 1: Execution Steps of D-TIME Emulator

```
1  Fetch Addr of essential functions in DLLs
2  success ← Attempt to open SCBC mutex
3  if success then
4      success ←Attempt to lock SCBC mutex
5      if sucess then
6          Open SCBC Semaphore
7          Comm Channel Addr ← Semaphore-counter
8      else
9          Wait for mutex (alternatively go to 42)
10         go to 4
11     end
12 else
13     Create and lock SCBC mutex
14     Attach shared memory
15     Create heap memory
16     Store heap memory addr. in shared memory
17     Create SCBC Semaphore
18     SCBC Semaphore counter ← Shared memory addr
19     Unlock SCBC Mutex
20 end
21 success ← False
22 i ← threadScheduler()
23 if i != -1 then
24     success ← Attempt to lock MTT[i].mutex
25     if success then
26         Next Chunk id ← MTT[i].nextChunk
27         Retrieve next chunk
28         Do re-locations in chunk
29         for r in {eax, ebx, ..., ebp, esp} do
30             r ← MTT[i].context.r
31         end
32         Execute chunk
33         MTT[i].nextChunk ← ebx
34         for r in {eax, ebx, ..., ebp, esp} do
35             MTT[i].context.r ← r
36         end
37         Release MTT[i].mutex
38     end
39 else
40     go to 52
41 end
42 for i ← 1 to No. of infected threads do
43     if MTT[i].threadId = currentThreadId then
44         MTT[i].queueCount--
45     else
46         if MTT[i].count < QueueThreashold then
47             Queue APC in MTT[i].threadId
48             MTT[i].queueCount++
49         end
50     end
51 end
52 Exit
```

SCBC is a convenient covert channel for the first emulator to broadcast the memory address to other emulators. SCBC along with the primary and secondary channels provide the communication channels for emulators to coordinate.

5.3 Emulator

The emulator is the most important component of D-TIME. It gets injected at random memory locations in infected processes and gets executed in the context of threads under these processes. Thus, the emulator should be written as a position independent function [48] to facilitate error-free execution. Our implementation of the emulator is a combination of assembly and C++ code. We build on from the emulator in [33].

While [33] has long living emulator threads executing chunks in a loop, our emulators are short-lived APCs which execute only one chunk in its execution lifetime. The steps involved in the emulator execution are briefly described in Algorithm 1. The rest of this section discusses the details of these steps.

Due to the position independent nature, the emulator will be unaware of the address of functions in Dynamic Linked Libraries (DLLs). Thus, the address of essential functions in the host process needs to be fetched at runtime (Line 1 in Algorithm 1). Every process in Windows stores all its metadata in a Process Environment Block (PEB) [15] structure. This data structure is available at the segment register offset `fs:[0x30]`. The PEB data structure contains the address of `PEB_LDR_DATA` [16] structure which contains a list of modules (DLLs) loaded by the process and their addresses. Once we have the base address of DLLs in the host process, the locations of functions in the DLLs can be easily retrieved.

Once the function addresses are obtained, the emulator determines if it is the first emulator to get executed in the process. The first emulator is supposed to initialise the communication channels for the process. To identify if the emulator is the first one, it checks for the existence of the SCBC for the process by attempting to open the mutex associated with the SCBC (refer Section 5.2) using the `OpenMutexA` API (Line 2). If the attempt succeeds (Line 5 to 11), it means that the channels were already initialised and the base address of the shared memory can be obtained from the semaphore counter. Otherwise, the initialisation is not done yet. In such a case, the emulator creates the mutex in a locked state (Line 13). The locked state of the mutex implies that the initialisation is under progress. Thus, other emulators if spawned during this time will have to wait for the initialisation to complete. With the mutex in locked state, the emulator initialises the communication channels by attaching the shared memory region and creating a heap memory region (Lines 14, 15). After initialising the communication channels, the base address of heap memory is stored in shared memory. Then the SCBC semaphore is created and the counter value is set to the base address of the shared memory. The SCBC mutex is then released allowing other emulators to read the address and connect to the channels. These are done in Lines 16 to 19 in Algorithm 1.

After connecting to the communication channels, it finds out the next malware thread to be scheduled by running the `threadScheduler` function (Line 22). `threadScheduler` schedules threads in the malware in a round robin fashion. If chunks of all threads are already under execution or if the malware has exited, the scheduler returns an invalid thread number (`-1`). Otherwise, a valid thread number is returned and the emulator can attempt to execute the next chunk. However, only one emulator will execute the chunk of a malware thread at a time. Thus, the emulator locks the mutex for the thread from Malware Thread Table (MTT) to earn the execution rights (refer Section 5.1) (Lines 24).

If the emulator gets to lock the mutex from MTT, it can now execute the next chunk. The emulator retrieves the identifier of the next chunk from MTT and copies the chunk into a section of the emulator's own body which otherwise is filled with `nop` instructions (Line 27). The copied chunk is then subjected to address resolution (Line 28). For example, the value of `jmp` targets are modified to suit the address range occupied. After address relocation, the context is set for the malware execution by copying the register values from context structure under MTT to respective registers (Lines 29 to 31). Then the emulator jumps to the first instruction of the relocated chunk to execute the chunk (Line 32). If the chunk execution demands a call to a function in DLLs, the emulator determines the address of the API on the fly and the call is made. After the execution of a chunk, the emulator obtains the identifier of the next chunk to be executed from `ebx` register (Line 33). The next chunk identifier, as well as the register values, are updated in MTT (Lines 34 to 36). Then the emulator releases the mutex to lose the chunk execution rights (Line 37).

As the last step, the emulator queues itself to other threads. The Infected Threads Table is looked up and the emulator is queued to threads devoid of pending emulators. The pending emulator counts are incremented along with queuing. Then the emulator decrements the pending emulator count for the current thread and exits (Lines 42 to 52).

5.4 Challenges in Implementing D-TIME

Distributing the malware execution creates a number of implementation challenges. For example, the stack, heap, files descriptors, and socket descriptors of the malware should be shared among emulators. Also, the consistency of process specific information provided by functions such as `GetCurrentProcessId` should be ensured. In this section, we discuss some of these important challenges.

Implementing Stack and Heap: We use an approach similar to [33] in order to implement the stack and heap for the malware. Stack is local to a thread while heap is a process local resource. Thus, we allocate one shared memory region for each malware thread and use it as stack and another shared memory region for the entire malware and use it as heap. Just prior to a chunk execution, the emulator will use the stack registers such as `esp` and `ebp` to point to appropriate locations in this created stack region. These details are stored in the MTT. To support the usage of heap memory, a compact version of standard heap APIs such as `malloc` and `HeapAlloc` are provided in the emulator. All calls to the heap related functions are redirected to these customised implementations.

Handling files and sockets: A file or a socket created by a process is only accessible to the specific process. This poses a challenge in D-TIME since a file or socket may be opened in one chunk and accessed (read or written) in another chunk. Thus a socket or file descriptor created should

Table 1: Statistics of chunks obtained in different chopping mechanisms. Size (in bytes) of the original malware, number of chunks, and average size (in bytes) of chunks for each three chopping algorithms. The BAST mode threshold is set to 16 bytes. All sizes are in bytes.

	Malware	Size	BBS		BAST		Paranoid	
			Count	Avg	Count	Avg	Count	Avg
M1	Offline-Keylogger	10,240	39	80.21	62	66.26	206	49.38
M2	Online-Keylogger	10,240	47	78.46	84	62.64	233	49.52
M3	Backdoor	10,240	96	90.15	165	70.26	450	52.42
M4	Screenshooter	11,264	34	99.29	114	59.52	400	47.02
M5	Window-Monitor	8,704	11	76.92	21	61.23	64	48.54
M6	Ransomeware	11,264	61	74.06	107	61.57	328	48.64

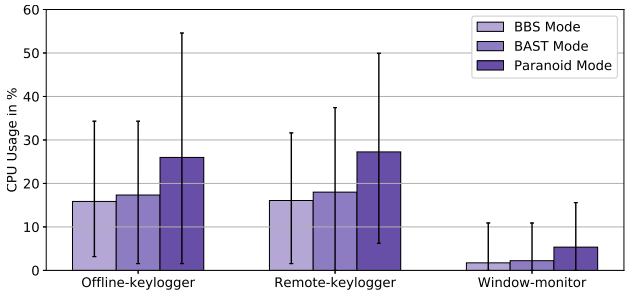

Figure 4: CPU usage of D-TIME infected processes at steady state for different chopping mechanisms. Four victim processes are used for each case.

be shared among all the emulators. The programming interface to files [36] and sockets [37] in Microsoft Windows are through handles. A handle opened by a process can be duplicated for other processes using the `DuplicateHandle` [8] API. As soon as a chunk execution creates a `handle`, the emulator duplicates the `handle` for all the infected processes. This information is then broadcasted over the primary channel. Emulators spawned in other processes can now use these duplicated `handles` to access the respective file or socket.

The above technique is useful in dealing with most of the Microsoft Windows APIs since most of them use handles. However, some rarely used handles like *search handle* returned by `FindFirstFile` [9] used to search through files in a directory cannot be duplicated. Similarly, Graphical Device Interface (GDI) [21] objects used to create graphical windows cannot be shared across processes. Custom implementations for APIs using these objects needs to be provided. Alternatively, the emulators can decide to execute all chunks requiring these APIs in the same process.

Providing process specific information: Any process specific information requested by chunks should be coherent across emulators. The information provided by APIs like `GetCurrentProcessId` [11], `GetCurrentDirectory` [10], etc. will be different for distinct victim processes. The coherency is maintained by redirecting these calls to custom APIs provided in the emulator. For example, D-TIME can decide to choose the process id of one of the infected process to be the `pid` of the malware. This id shall be returned when the chunk attempts to call `GetCurrentProcessId`.

6 Evaluation and Results

In this section, we discuss the results of our experiments and analysis of D-TIME. We selected six different malware samples for demonstrating the Proof of Concept (PoC). These samples capture the common behaviour of malware as described in [22]. The sample set includes an offline key-logger (M1) which logs keystrokes in a file and a remote key-logger (M2) which sends keystrokes over the network; a backdoor (M3), which provides command line control for the remote attacker; a malware that periodically captures screenshots (M4); a malware that monitor the system and kills applications like TaskManager (M5), and a ransomware (M6). All the malware were written in C++ and compiled for 32-bit Intel x86 platform. The executables were obfuscated with D-TIME and tested on Microsoft Windows 10 Operating System.

The chunks obtained after obfuscation were injected into common software such as Chrome (version 74.0.3729.131), Skype(version 8.45.0.41), VLC (version 3.0.6), Acrobat Reader (version 19.10.20099.322322), Opera (version 58.0.3135.132) and Calculator (version 1803). Since the emulator was written and compiled for a 32 bit architecture, 32 bit versions of the software were used as targets.

The key to remain stealthy is to minimally influence the normal behaviour of the targets. We therefore evaluated D-TIME for its affect on the performance of the system under various configurations. We also tested D-TIME against the current state-of-the-art detection techniques using popular anti-virus software. This section provides details of the evaluation.

6.1 Effect of different Splitting Algorithms

D-TIME works with different splitting algorithms: BBS, BAST, and Paranoid mode, described in Section 4.1. Table 1 shows the statistics of chunks in the 6 PoCs for different splitting mechanisms. The average chunk size is highest in BBS mode and lowest in Paranoid mode. Each splitting mechanism has different performance impacts. These variations are shown in Figure 4. Evidently, chopping mechanisms with higher granularity causes higher performance impact. When the chunks are small, the total number of chunks is higher. Thus it demands more emulator executions, increasing the execution overhead.

6.2 Performance Impact

A malware can be considered to be more stealthy if it only has a minimal impact on CPU usage. We measured the CPU usage of victim processes under D-TIME infection. The test was conducted in multiple configurations and the results are

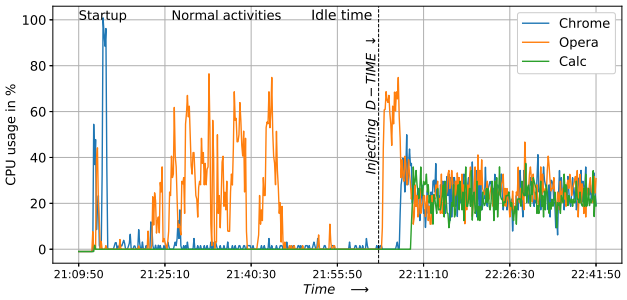

Figure 5: CPU usage vs Time graph for processes during startup, normal activities, when idle and while infected by D-TIME obfuscated keylogger. In each victim process, all threads up to a maximum of 16 threads were infected.

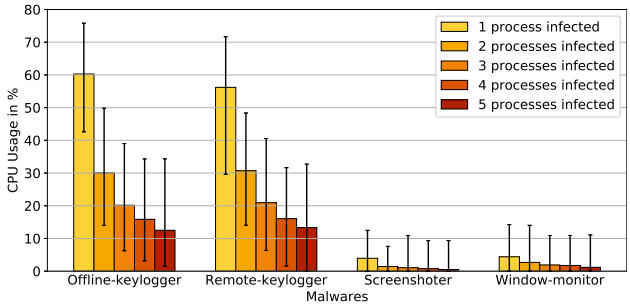

Figure 6: CPU usage of D-TIME infected processes at steady state for a varying number of victim processes. In each victim process, all the threads up to a maximum of 16 threads were infected.

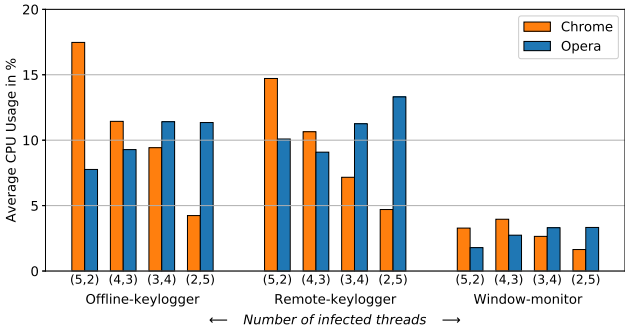

Figure 7: CPU usage of D-TIME infected processes with four different thread distributions. 7 threads were infected in total across Chrome and Opera in (5,2), (4,3), (3,4) and (2,5) configurations. For example, 5 threads in Chrome and 2 threads in Opera are infected in the first configuration.

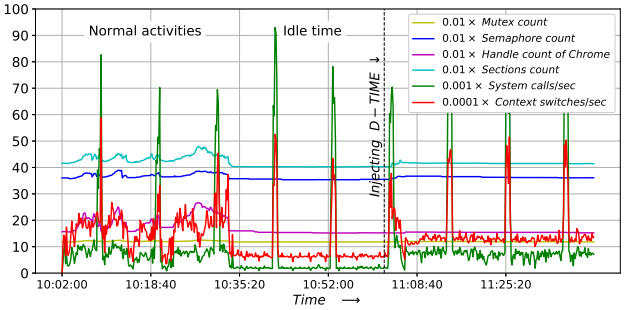

Figure 8: Variation of various counter values upon D-TIME infection of a target. System calls/sec and context switches/sec can be seen to have a considerable deviation.

given in Figures 5, 6 and 7. In all configurations, the victim processes were observed to continue to serve their benign purpose even after D-TIME infection. Figure 5 shows the CPU usage of processes on startup, during normal activities, when idle and when infected by D-TIME. The CPU usage of the victim processes are found to settle to an average of 20% after a few initial transients.

Figure 6 shows the variation in the average CPU usage of victims for different number of infected processes. Clearly, the CPU usage of a victim reduces as the number of infected processes increase. It is heavily depended on the malware chunks as well. For example, the keylogger, which loops on tiny chunks demand a higher frequency of emulator executions causing a higher CPU usage compared to Screenshooter, which is looping over bigger chunks. This trend is much more clear in the discussions of Section 6.1.

Figure 7 shows the average CPU usage of two victim processes for different distributions of infected threads. We infected 7 threads across Chrome and Opera distributing the threads as (5, 2), (4, 3), (3, 4), and (2, 5) respectively in each process. For example, 5 threads in Chrome and 2 threads in Opera are infected in the first configuration. The distribution of CPU usage of processes is found to be similar to the distribution of the number of threads. Thus, varying the number of threads is a practical method of manipulating the CPU usage of D-TIME.

Table 2: Anti-virus software used for testing D-TIME.

Antivirus	Version
BitDefender	23.0.22.97
Norton	22.17.0.183
Kaspersky	19.0.0.1088
WEBROOT	9.0.24.49
McAfee	17.8
ESET	12.1.34.0
Avast	19.4.2374
AVG	19.4.3089
Windows Defender	4.18.1903.4
Avira	1.2.133.21088

6.3 Detectability

To test detectability, we executed D-TIME against 10 popular anti-virus software in the industry, listed in Table 2. The test was conducted for different configurations of infected processes (1 to 6 processes) and splitting mechanisms (BBS, BAST and Paranoid Modes). No anti-virus was able to detect D-TIME on any configuration.

Performance Counter based malware detection is an emerg-

ing detection technology [49]. Commonly used Performance Counters such as cache hits, page faults etc. [28] did not show any considerable variations upon D-TIME infection. However, we found the number of Context Switches and System calls to show considerable deviations under D-TIME infection. Figure 8 shows the variation of these counters when infected by D-TIME. During our tests, to isolate the variations, only D-TIME was kept running after the infection. The variations can be easily understood from D-TIME design which is heavily depends on APC executions. The user-mode APC executions cause a lot of context switches (refer to Appendix A). This observation may be used as a means to detect D-TIME.

7 Countermeasures

In this section we discuss detection and removal techniques for D-TIME. We also suggest countermeasures to prevent such attacks.

Prevention Techniques: The best way to prevent D-TIME is by improving the Operating System design to prevent the attack. D-TIME is built upon the Microsoft Windows feature called APC and a covert channel named SCBC. Adding restrictions to these features can help prevent the attack.

- *Queue Privilege Class (QPC)*: We recommend that every thread has a list called QPC, containing the identifiers of threads which can queue APCs in its APC-Queue. This restricts arbitrary threads from queuing APC on the target.
- *Limiting Recursive Queuing*: The D-TIME framework is heavily depended on the self-regeneration of emulators by queuing themselves as APCs in other threads recursively. Such recursive APC queuing is unlikely to be legitimate thus restricting the maximum depth of recursive queuing of APCs can prevent the attack.
- *Restricting handle duplication*: The Operating System shall restrict inter-process duplication as well as sharing of handles. This can considerably restrict D-TIME and all other forms of distributed malware.
- *Reducing covert channels*: SCBC exist in the system since the semaphore counter value can be read by the clients. The ability to make repeated wait and release calls helps the attacker to reuse a semaphore for multiple broadcasts. By restricting the access to semaphore like resources, covert channels like SCBC can be prevented.

Detection Techniques: Detecting D-TIME using behavioural and heuristic methods is intricate if not impractical. Such an analysis would require the anti-virus software to find connections between D-TIME executions across processes and threads. This is a hard problem due to the small size of the chunks executed.

A possible way of detecting D-TIME is by detecting its communications channels. However, D-TIME design is flexible to accommodate new channel designs. D-TIME can also change the communications channels dynamically at runtime. This can make the detection of side-channels a more challenging problem.

Another approach of detecting D-TIME would be by tracking CPU usage by processes and performance counters in the system. Typically, only a very few processes (usually one or two) will have a considerably high CPU usage in the operating system at a time. We notice that almost all the D-TIME infected processes have similar CPU usage pattern. This similarity of CPU usage can be used as an indication of D-TIME infection. Similarly, the rate of context switches shows a sudden increase which remains steady under D-TIME infection. This too may be used as a means of detecting D-TIME.

Removal of D-TIME: Removing D-TIME from an infected system is difficult. The defender should remove all the emulators or communication channels simultaneously to take out D-TIME. A single emulator and a channel left in the system is sufficient to bring D-TIME back to its original state due to the regenerating nature of emulators.

8 Conclusion

This paper describes D-TIME, a malware execution framework that can evade behavioural and heuristic runtime detection by executing small chunks of malware injected in benign threads in a distributed fashion. A well coordinated execution of these chunks helps achieve the malicious action.

D-TIME makes use of the Asynchronous Procedure Call (APC) feature present in Microsoft Windows, to execute chunks in arbitrary threads. It also uses the value stored in a semaphore to create a covert channel between the various chunk executions.

We evaluated D-TIME for its stealth with 10 state-of-the-art anti-virus software, CPU usage, and Performance Counters. All ten anti-virus software were unable to detect D-TIME. We seen an average increase in the CPU utilization by 20% when a D-TIME infected process executes.

While most Performance Counters evaluated were also unable to capture any specific runtime signature of D-TIME, there was a slight increase in the number of context switches. A heuristic based queuing protocol may be devised to reduce the number of unused emulators leading to a lower rate of context switches and lesser impact on the CPU usage. Similarly, minimizing the rate at which emulators are spawned would also achieve the same effect.

The success of D-TIME highlights the need for stronger malware detection techniques that should cater to distributed malware. It also emphasizes the need for Operating Systems design to be more aligned to security polices like the principle of minimum disclosure.

Availability

The source code of our PoC is available at https://github.com/JithinPavithran/d-time.

References

[1] Five reasons to use vmprotect. https://vmpsoft.com/. Accessed: 2019-05-14.

[2] Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques. https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process. Accessed: 2019-05-17.

[3] *J. Comput. Secur.*, 6(3), 1998.

[4] A virus detection scheme based on features of control flow graph. In *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*, pages 943–947, Aug 2011.

[5] Alertable I/O. https://docs.microsoft.com/en-us/windows/desktop/fileio/alertable-i-o, 2018. Accessed: 2019-05-26.

[6] CloseHandle function. https://docs.microsoft.com/en-gb/windows/desktop/api/handleapi/nf-handleapi-closehandle, 2018. Accessed: 2019-05-26.

[7] CreateSemaphoreA function. https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-createsemaphorea, 2018. Accessed: 2019-05-26.

[8] DuplicateHandle function. https://docs.microsoft.com/en-us/windows/desktop/api/handleapi/nf-handleapi-duplicatehandle, 2018. Accessed: 2019-05-26.

[9] FindFirstFileA function. https://docs.microsoft.com/en-us/windows/desktop/api/fileapi/nf-fileapi-findfirstfilea, 2018. Accessed: 2019-05-26.

[10] GetCurrentDirectory function. https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-getcurrentdirectory, 2018. Accessed: 2019-05-26.

[11] GetCurrentProcessId function. https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentprocessid, 2018. Accessed: 2019-05-26.

[12] Interprocess Communications. https://docs.microsoft.com/en-us/windows/desktop/ipc/interprocess-communications, 2018. Accessed: 2019-05-26.

[13] OpenFileMappingA function. https://docs.microsoft.com/en-us/windows/desktop/api/winbase/nf-winbase-openfilemappinga, 2018. Accessed: 2019-05-26.

[14] OpenSemaphore function. https://msdn.microsoft.com/en-us/windows/desktop/ms684326, 2018. Accessed: 2019-05-26.

[15] PEB Structure. https://docs.microsoft.com/en-us/windows/desktop/api/winternl/ns-winternl-_peb, 2018. Accessed: 2019-05-26.

[16] PEB_LDR_DATA structure. https://docs.microsoft.com/en-gb/windows/desktop/api/winternl/ns-winternl-_peb_ldr_data, 2018. Accessed: 2019-05-26.

[17] QueueUserAPC function. https://docs.microsoft.com/en-gb/windows/desktop/api/processthreadsapi/nf-processthreadsapi-queueuserapc, 2018. Accessed: 2019-05-26.

[18] ReleaseSemaphore function. https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-releasesemaphore, 2018. Accessed: 2019-05-26.

[19] WaitForMultipleObjects function. https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitformultipleobjects, 2018. Accessed: 2019-05-26.

[20] WaitForSingleObject function. https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject, 2018. Accessed: 2019-05-26.

[21] Windows GDI. https://docs.microsoft.com/en-us/windows/desktop/gdi/windows-gdi, 2018. Accessed: 2019-05-26.

[22] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[23] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh. A survey on heuristic malware detection techniques. In *The 5th Conference on Information and Knowledge Technology*, pages 113–120, May 2013.

[24] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*, 2007.

[25] Jean-Marie Borello, Éric Filiol, and Ludovic Mé. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in Computer Virology*, 6(3):277–287, Aug 2010.

[26] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 129–143. Springer, 2006.

[27] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.

[28] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 559–570. ACM, 2013.

[29] L. Dessau and R. B. Kajberg. An analysis of x86 single-instruction compiling as an obfuscation technique. Master's thesis, Technical University of Denmark, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Building 324, DK-2800 Kgs. Lyngby, Denmark, compute@compute.dtu.dk, 2016. DTU supervisor: Christian D. Jensen, cdje@dtu.dk, DTU Compute.

[30] Stephen Dolan. mov is turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[31] Christopher Domas. Break Me00 The MoVfuscator Turning mov into a soul crushing RE nightmare Christopher Domas. https://www.youtube.com/watch?v=R7EEoWg6Ekk, 2015. Accessed: 2019-05-26.

[32] Gheorghe Hăjmăşan, Alexandra Mondoc, Radu Portase, and Octavian CreŢ. Evasive malware detection using groups of processes. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors, *ICT Systems Security and Privacy Protection*, pages 32–45, Cham, 2017. Springer International Publishing.

[33] Kyriakos K. Ispoglou and Mathias Payer. malWASH: Washing Malware to Evade Dynamic Analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, 2016. USENIX Association.

[34] S. Jana and V. Shmatikov. Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94, May 2012.

[35] K. Jeong and H. Lee. Code graph for malware detection. In *2008 International Conference on Information Networking*, pages 1–5, Jan 2008.

[36] Michael Satran John Kennedy. File Management Functions. https://docs.microsoft.com/en-us/windows/desktop/fileio/file-management-functions, 2018. Accessed: 2019-05-26.

[37] Michael Satran John Kennedy. Getting Started with Winsock. https://docs.microsoft.com/en-us/windows/desktop/winsock/getting-started-with-winsock, 2018. Accessed: 2019-05-26.

[38] Michael Satran John Kennedy. Semaphore Objects. https://docs.microsoft.com/en-us/windows/desktop/ipc/interprocess-communications, 2018. Accessed: 2019-05-26.

[39] B. Min and V. Varadharajan. Design and analysis of a new feature-distributed malware. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 457–464, Sep. 2014.

[40] Jon Oberheide, Michael Bailey, and Farnam Jahanian. Polypack: an automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, pages 9–9. USENIX Association, 2009.

[41] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Evolution of computer virus concealment and anti-virus techniques: a short survey. *arXiv preprint arXiv:1104.1070*, 2011.

[42] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.

[43] Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[44] Neha Runwal, Richard M Low, and Mark Stamp. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8(1-2):37–52, 2012.

[45] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[46] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification techniques on opcode patterns. *Security Informatics*, 1(1):1, 2012.

[47] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[48] Michael Sikorski and Andrew Honig. *Practical malware analysis*, chapter 19, pages 408–409. No Starch Press,us, 2012.

[49] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 109–129, Cham, 2014. Springer International Publishing.

[50] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, Dec 2006.

[51] Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. Imds: Intelligent malware detection system. pages 1043–1047, 01 2007.

[52] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

Appendices

A Reverse Engineering User-mode APC

Though APC is a widely used feature in Microsoft Windows, very less documentation is available on its internals. The internals is complicated and requires a deep knowledge of the Microsoft Windows internals to understand them. In this section, we discuss a high level overview of how User-mode APCs are dispatched by abstracting out the details for an easy understanding.

User-mode APCs are triggered when the target thread enters *Alertable Wait State*, which internally results in a call to System Service Dispatcher (SSD). SSD is an interface in Microsoft Windows for many system services. Its job is to load `eax` register with the identifier of the requested service and switch to ring 0 by means of `sysenter` instruction. `sysenter` transfers the control to `nt!KiFastCallEntry` which in turn executes the requested service. Once the service execution is completed, before resuming to ring 3, `nt!KiFastCallEntry` checks for any pending User-mode APC that is queued. If there are pending User-mode APCs, `nt!KiDeliverApc` is called to deliver these APCs. After doing the essential queue management steps such as dequeuing the first APC, `nt!KiDeliverApc` takes the help of `nt!KiInitializeUserApc` to modify the user-mode stack, user-mode context and ring 0 `eip` such that the thread gets hijacked to call `KiUserApcDispatcher` when it returns to ring 3. When the thread resumes in ring 3, it executes `KiUserApcDispatcher` which triggers the User-mode APC function. After the execution, `KiUserApcDispatcher` calls `ntdll!NtContinue` to set the user-mode context and pre-APC `eip` to resume the execution from wait call which resulted in the *Alertable Wait State*. `ntdll!NtContinue` is actually implemented as an executive service which results in a call to `nt!NtContinue` which checks the APC-Queue for any User-mode APC and calls `nt!KiDeliverApc` again to deliver the next APC. When there are no more APCs, `nt!NtContinue` transfer the control back to the normal mode to resume execution from the pre-APC instruction.

The repeated transfer of control between ring 0 and ring 3 causes a considerable number of context switches during the delivery of user-mode APC. The kernel-mode APCs complete the delivery in ring 0 and does not cause the overhead of context switches. However, administrative privileges are required to queue kernel-mode APCs.