Pure User Mode Deterministic Replay on Windows

Atle Nærum Eriksen

Thesis submitted for the degree of
Master in Informatics: programming and networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2017

Pure User Mode Deterministic Replay on Windows

Atle Nærum Eriksen

August 1, 2017

Abstract

The ability to record and replay program executions has many interesting applications such as debugging in the backwards direction, discovering and fixing the source of non-deterministic bugs and data races and retracing the steps of a system intrusion. Unfortunately, the power of deterministic replay tools is underutilized by the general public as the tools are either too difficult to deploy or unable to fulfill the performance and log size requirements of the user. As it happens, the majority of the research has been aimed at implementing such tools for Linux, and other platforms, including Windows, have mostly been neglected.

In this thesis we look at whether it is possible to implement a deterministic replay tool for the Windows platform that is easily deployable (user mode only without operating system support – this entails no OS modifications or drivers), can record all system calls and their side-effects (even if unknown), works on large programs (1 GB+ RAM), and has a high recording performance (≈2x slowdown).

We found that the challenges deterministic replay tools are facing in user mode are exacerbated on Windows due to a lack of documentation and a more restrictive API. Despite this we came up with a design proposal that solves all the problems necessary to implement a deterministic replay tool that satisfies all our requirements for the Windows platform.

We present novel techniques to record thread scheduling and non-deterministic instructions. We also describe in detail how to recreate the address space of a recorded program in which code can be executed and access resources directly without instrumentation or modifications just like in the original program. An alternative novel approach to this technique is also suggested. None of the methods rely on operating system support.

Although the design proposal remains theoretical we have implemented two partial prototypes that were used to experiment on a small dummy program. Our findings show that it is reasonable to expect a recording slowdown on real programs in the range of 1-5x that will stay consistent even on programs with high memory usage. Regardless, the results are not conclusive and should be taken with a grain of salt.

Acknowledgements

I would like to express my sincere gratitude to my advisors Trond Arne Sørby and Pål Halvorsen for giving me the opportunity to work on this amazing project. Considering the circumstances this was not a given, and I wanted to let you know that I am eternally grateful, in particular for giving me the freedom to run my own show. I experienced significant growth as a programmer and I consider the project to be one of the most difficult I have ever taken on – and it was totally worth it! I appreciated all of your input even though I did not have the time to act upon all of it. This thesis helped me grow as a person in unexpected ways and I thank you for this life-changing opportunity.

I also want to thank Duncan Ogilvie, known in the community as mrexodia, for starting and maintaining the x64dbg project, the debugger I have been using while implementing this project. Let me just say this is a seriously good debugger. I have spent, or should I say "time wasted debugging", literally weeks of my life staring at assembly code in the time I have been working on this thesis, and the other debuggers I would normally use just kept crashing over and over due to the abnormal behavior of my program (think malware), but x64dbg failed me only a few times in the timespan of a whole year. I can guarantee you this thesis would not exist without this debugger. A shout-out to all the developers on this project!

I dedicate this thesis to Lisa, Erika and Miku. Thank you for your never-ending love and support.

Contents

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Computer programs are deterministic; if given the same inputs on every execution, they will always produce the same outputs. This statement remains true even if we consider sources of randomness such as *rand()* and *rdtsc*, because the condition states: *given the same inputs* – ergo if *rand()* returns 42 twice the program will output the exact same result both times. The *skeleton* of the program remains static after compile-time and it is only the inputs that can change, and sources of randomness are considered as inputs to the program for the reason that they could not be predicted and optimized at compile-time.

The property of determinism helps us in formally evaluating a program, because the sequence of steps taken inside the program and the resulting outcome may be predicted based on the given inputs. Unfortunately, this is not the case in practice. Any program of reasonable size will contain sources of non-determinism such as user input, network- and disk I/O, OS scheduling, and current-time-retrieval. Any kind of program input that can not be predicted beforehand can be classified as *non-deterministic*. Even in the case where no such inputs have been added by the programmer it may have been added by the compiler.

If the program's execution path can not be predicted, it can be very difficult to formally assess the correctness, robustness, safety, and resource usage of the program as required by its specification. Another problem with non-determinism is determining the cause of a program crash, because if the

program is restarted in a debugger all the non-deterministic events must be reproduced to reach the point of the crash, something which in some cases is simply not possible.

To help remedy these issues the research area of *deterministic replay* emerged, which focuses on recording and replaying program executions deterministically. This entails logging all non-deterministic events and later reproducing them at the exact location at which they occurred in the first program run while running the program a second time. The result is that both program runs, and any other future replay runs, take identical execution paths, and can therefore be analyzed without fear of disruption by non-determinism. The tools used to achieve deterministic replay are called *deterministic replay tools*.

Some deterministic replay tools offer *time-travel debugging* functionality. Time-travel debugging is the concept of making the debugger able to work in the backwards direction in addition to its normal function in the forward direction [53]. Often when debugging the developer will stumble upon a piece of information that is not useful in the present, but would have been useful earlier in the debugging process. Time-travel debugging solves this problem by letting the developer "go back in time". The alternative would be to re-run the program being debugged and hope that the acquired information has not been outdated due to non-determinism. Difficult debugging scenarios, such as identifying uninitialized variables or memory leaks, become much simpler to analyze when the program execution can be debugged iteratively without having to restart the program (only the replay), as all events are pulled from the log [80]. Some deterministic replay tools also offer a programmable API which allows time-travel debugging to be extended to the scripting realm for automated analysis, allowing for extended use of already known techniques such as fuzzing, symbolic execution, dynamic slicing and taint analysis.

1.2 Problem Statement

There exist a multitude of deterministic replay tools already which offer satisfying results. The problem is that very few of them operate in user mode without operating system support, nor are they necessarily easily accessible to the average developer. Additionally, the vast majority of such tools have been created for Linux; the number of deterministic replay tools we were able to find for the Windows platform we could count on one hand, and they all

relied on operating system support. The reason for this is that Windows is not as flexible as Linux when it comes to the needs of deterministic replay.

By relying on help from the operating system to perform certain tasks the deterministic replay tool is harder to deploy because the operating system kernel must be modified or a kernel driver must be used, none of which are trivial to set up. Operating system support offers improved performance and control, but to the average user it is only a hindrance.

We believe that the possibilities of deterministic replay are too important to remain unknown to most of the world's developers and that by offering a solution which is easy to deploy and also works on Windows, a platform with a very large number of users and developers, that this would enable the average developer to improve their workflow with significant effect. This brings us to our first question:

> *Considering the lack of deterministic replay tools on Windows and the limits of the platform compared to Linux in the context of deterministic replay, is it possible to implement a deterministic replay tool in user mode without operating system support for the Windows platform?*

It is typical for deterministic replay tools to monitor the system call interface between the user program and the operating system in order to log the system calls. The system calls are handled individually according to their documentation. This limits the deterministic replay tool in that it can only support the system calls that have been explicitly whitelisted, and the tool may not work with programs that happen to use unsupported system calls.

On Windows, system calls are considered an implementation detail that should not be touched by developers, and as a result there is a lack of documentation. This is also true for user space internals; Microsoft has purposely avoided to publicly document these data structures so that they can be updated without having to support backwards compatibility – the WinAPI is there to shield these data structures and support this backwards compatibility. It is therefore difficult to determine whether a system call had any undocumented side-effects that should have been recorded, because we cannot know if the system call updated hidden internal structures in the address space. In conjunction with the whitelisting issue we therefore ask the next question:

> *Is it possible to efficiently record arbitrary system calls and their side-effects in an automatic manner without manual handling so*

that documentation and knowledge of user space internals is not necessary, with no operating system support?

Performing deterministic replay in user mode without operating system support implies a significant performance penalty, because recording non-deterministic events without help from the operating system is not always straightforward and therefore time-consuming. A performance slowdown must be expected in general when recording a program, and for many programs this is acceptable, but for time-sensitive tasks such as network communication where the server may disconnect the client, time is of the essence. Additionally, programs with a large memory footprint, e.g., games, may suffer from additional performance degradation due to the recorder's poor ability to scale. Without operating system support these problems are likely to be worsened.

For these reasons, we decided to pursue the most speed-efficient implementation possible with regards to raw performance and scalability. This would allow our deterministic replay tool to be able to record the aforementioned kind of programs. Additionally, as recording performance is a pronounced problem for user mode deterministic replay tools we wanted to simply see how far we could push performance to its limit since no previous research appears to have put in a direct effort of optimizing past a certain point (i.e., they stopped when reaching an acceptable level of performance).

As a result of this decision we had to let other concerns come second, such as log size. It is worth noting that replay performance is not as important as recording performance since replays may be carried out at any time and do not require human interaction, thus there is less incentive to pursue the same level of raw speed. Replay performance is therefore not of importance. The third question we ask is:

Is it possible to achieve a recording performance of ≈2x slowdown, which also scales to large programs (1 GB+ RAM), with no operating system support?

To answer these questions we developed a design proposal detailing how a deterministic replay tool could be implemented which satisfies all the mentioned criteria. In order to help us make educated decisions during the design process we implemented a recording component prototype and a replayer component prototype. By using both components together we have a deterministic replay tool. The design proposal was given the name *Dragoon*.

1.3 Limitations and Scope

We limit the scope of Dragoon to unmodified Windows 7 64-bit on the x86 architecture. It should record unmodified 32-bit user mode programs (WOW64), in binary form (Portable Executable), that run on a single processor (parallel programs will still be recorded, but their parallelization will be eliminated). Recording should happen at instruction granularity, and it should be possible to revisit every instruction executed and every memory access made by the recorded program. The replay component should be implementable on any platform for flexibility (e.g., record on Windows but take advantage of all the additional analysis tools on Linux during replay), and should offer a generic API to support scripted automated analysis and manual time-travel debugging in the case of debugger integration. Dragoon will not focus on issues such as data race elimination or crash dump analysis (which is common for deterministic replay tools), and is intended for reverse engineering and time-travel debugging. The recording overhead should preferably remain below 2x the original execution speed, and other research employing similar techniques to Dragoon's have shown this to be achievable ([82], [3], [9]). It should be possible to record a program from start to finish in order to search for early variable initializations, but also to record only a certain time period, similar to the start- and stop buttons on a recording device. Lastly, to support the recording of interpreted languages and malware, it should be possible to record self-modifying code.

Deterministic replay tools implemented in user mode without operating system support have previously been demonstrated to be possible, but they come with unreasonable caveats and would therefore not fit our requirements. Examples include requiring the tool to be compiled or linked in with the original program instead of recording unmodified pre-compiled binaries, recording at a higher granularity than instruction granularity, not being able to properly record self-modifying code, or the recording overhead is too high (PinPlay [84] is an extreme example with 100x-200x slowdown compared to native execution). We will in this thesis look for a solution that has the least amount of unreasonable trade-offs as much as is possible.

The most explicit limitation of Dragoon is that any action not taken by the program being recorded itself will not be recorded by Dragoon. For instance, changes to shared memory by external processes and other interference from the outside world falls in under this category.

Due to the size of the project we do not expect to acquire any benchmarks

at this stage from real programs, and limit our evaluation of Dragoon to a small dummy program. The evaluation is intended to display the direction in which the project is headed and help us in making educated decisions about how Dragoon may potentially perform on real programs in the future.

1.4 Research Method

The research presented in this thesis follows the design paradigm described by the ACM Task Force in *Computing as a Discipline* [16]. The steps involved in this paradigm are to state requirements and specifications, then design and implement the system, and finally test the system.

1.5 Main Contributions

In this thesis we have shown that it is possible to implement a deterministic replay tool in user mode for the Windows platform which does not rely on operating system support, and our design is generic enough to be transferred to most operating systems with some exceptions.

We present a design proposal for a complete deterministic replay tool and the two prototypes that were used for experimentation. The prototypes are incomplete and therefore do not fully implement all the considered design suggestions, but they have helped us in gaining an intuition for assessing how valid our claims may be. This is further supported by tangible test results that, despite not being conclusive, show promise in that our design may hold up in practice.

We have successfully managed to attain a recording performance which is relatively close to our requirement, and in addition we show that it is possible to achieve automatic recording of arbitrary system calls while capturing their unknown side-effects without having access to their documentation nor operating system support. Both results still need improvement, but considering our rather strict requirements the outcome is positively satisfying.

It is, however, very important to note that all tests were performed on a small dummy program intended to help predict the needs of Dragoon for further development; that is to say that Dragoon is currently not mature enough to be run on real programs, but is very close as most of the infrastructure has been implemented. Our efforts were focused on optimizing and

ensuring that Dragoon could fulfill the specified requirements adequately, as the most important goal of this thesis is to push recording performance to its limit, not just to be able to record, which is a much simpler matter. This proved to be quite challenging and priorizations therefore had to be made. Regardless, all the facets of implementing a deterministic replay tool have been covered by the design proposal despite not having been fully validated by the prototype.

We also describe multiple ways to implement a replay component and show in detail how to implement the one with the highest implementation cost, which is to clear out a new process's address space completely so that it can be recreated to simulate the address space of the recorded process. This is not a new approach, but we provide code which shows how the implementation is to be carried out, something which we do not believe have been presented in earlier work. Considering the high implementation cost of this method we deem it an important contribution as it is likely to help other researchers in avoiding the hardships we had to go through. Additionally, our solution does not require operating system support, and there is no indication that previous research has been able to fulfill this criterion, and will as a result have a higher deployment cost, thereby possibly making our approach the first of its kind on Windows.

We then show how thread scheduling can be simulated during the replay phase without the use of software or hardware counters, nor operating system support, which are the traditional ways of implementing thread scheduling simulation.

Lastly, we present, to the best of our knowledge, three novel contributions to the field.

The first is the ability to record thread scheduling in a multi-threaded program without operating system support or program modifications of any kind, thus allowing the operating system scheduler to still remain in control and make potentially optimal scheduling decisions. The Debugging API is not being used, which is a well-known traditional approach. We expect our solution to be on par with traditional solutions with regards to performance.

The second contribution is the ability to record common non-deterministic instructions (rdtsc, cpuid, etc.) without operating system support or dynamic code instrumentation. To our knowledge this has not previously been possible. It is likely our solution can be extended to other more rare non-deterministic instructions as well.

The third contribution is an alternative approach to the address space

recreation technique already presented with one key difference. It recreates the recorded address space in the same physical address space as the replay component and ensures that the deterministic replay of the logged code has all its memory accesses redirected to the recreated address space without perturbing the replayer's. This is opposed to recreating the address space in its own isolated process, something which on Windows proved to come with strict limitations that our proposed alternative bypasses in trade of lower performance.

1.6 Outline

The thesis is structured as follows:

Chapter 2: Technical Background
We begin by describing technical concepts required to understand this thesis. First we give an overview of the functioning of Windows, both documented and undocumented. Second we introduce topics from the field of program analysis such as hooking, instrumentation and emulation.

Chapter 3: Overview of Deterministic Replay
In the next chapter we introduce the research area of deterministic replay. We learn why non-determinism is a problem, familiarize ourselves with the taxonomy of deterministic replay with some mentions of the current state of the art, and get acquainted with the applications of deterministic replay.

Chapter 4: Deterministic Replay in User Mode
In this chapter we narrow our focus to user mode. We discuss how the user mode term can be misleading and gradually constrain it to have a more clear definition. The central topic of operating system support is discussed and we describe why Windows lacks flexibility to lend operating system support. Next, the concept of instrumentation is expanded upon in the context of deterministic replay, showcasing its importance. Lastly, we discuss all the different challenges deterministic replay tools face in user mode.

Chapter 5: Dragoon: A Deterministic Replay Tool Design Proposal for Windows

Once all the theory has been discussed, it is time to look at our proposed design. This chapter describes the design requirements and limitations, gives an overview of the system components, then expands upon the overview by discussing both the recording- and the replay component in detail.

Chapter 6: Evaluation
In this chapter, we take a closer look at the tangible results we have managed to gather from the unfinished prototypes. Memory usage, log size and performance is discussed, and each section is concluded by presenting definitive results.

Chapter 7: Conclusion
Finally, we conclude the thesis by summarizing our work and then discuss the potential for future research.

Chapter 2

Technical Background

In this chapter we will briefly introduce technical concepts that are of importance to this thesis. Every concept is described in the context of Windows 7 64-bit WOW64 on the x86 architecture, and we are only concerned about what is visible in user mode. The reader is assumed to already be familiar with operating system basics and x86 assembly.

2.1 Windows Operating System Background

We will in this section describe how the Windows operating system conducts itself in user mode and introduce technical details that are not officially documented, yet necessary to create a functioning design later in this thesis.

2.1.1 Processes and Threads

Processes

Every invocation of a user program runs in its own individual execution environment known as a *process*. Changes within a process will not interfere with other processes, even if they are all running the same program. This makes it possible to run the same program multiple times and receive different outcomes for each one.

The process environment can be thought of as a container. In user mode, a process consists of virtual memory only; any information used by the program resides here. The virtual memory of a process is referred to as the *address space* of the process (virtual memory will be discussed further in Section

2.1.3). In the context of a user mode process, the address space is also referred to as *user space* in contrast to *kernel space*. When a process is started, the program's code and data are copied into the process's address space along with other hidden information used by Windows. Windows then identifies the start location of the program code and instantiates the program's main thread at that location to achieve execution.

Threads

A *thread* is a construct which executes processor instructions within a process. Multiple threads may exist at the same time, performing different, independent jobs for a process. For example, a GUI program would use one thread for the user interface and another to perform calculations, otherwise every action taken by the user would freeze the user interface while the action was carried out. At least one thread is required at minimum to be able to run a program, because processes are only able to contain program code, not execute it. Every thread has its own settings and processor register context so that different threads do not interfere with each others data.

Thread Scheduling The Windows scheduler decides when threads are allowed to run, and it is the developer's responsibility to ensure that shared resources (the address space is such a shared resource) are accessed in a safe manner, otherwise data races may occur. If the computer has more than one processor core, different threads may be scheduled on different cores to improve performance. If there is only one processor, threads will run one at a time in turn; it is impossible for more than one thread to run at a time per processor. Threads still remain viable despite this; see the GUI program example above. Individual threads may be moved to different processor cores at will, may be suspended and resumed or may be given different priorities so that the scheduler is more likely to run the important threads first.

Thread Stacks Each thread is given an allocation of memory in the address space known as a *thread stack*. The thread uses this memory as scratch space for temporary information that can not, or do not semantically, fit in the thread's registers. The most common usage is to place function call frames on the stack and later remove them as the function is exited by the thread. As every thread has its own stack, it is therefore possible to keep track of deep function call hierarchies even when switching between different

threads. Programs running under WOW64 (WOW64 is described in Section 2.1.5) will have two thread stacks: one 32-bit stack and one 64-bit stack.

Thread Life Cycle Every thread takes its first steps at the start address of the *LdrInitializeThunk* function in the *ntdll.dll* system module (more on modules in Section 2.1.4) [48, 107, 19]. This is a special function known as a *user mode callback* (see Section 2.1.7) whose purpose is to initialize the thread (or the process, if the thread happens to be the first to ever execute). Examples of what happens during this initialization process is that exception handling is set up for the thread, so that if no exception handlers have been added by the developer, there will still be a top-level handler (more on exception handling in Section 2.1.8) [48, 85], *DllMain* of all DLLs are invoked with parameter *DLL_THREAD_ATTACH* to signal the creation of the new thread, and the thread's TLS callbacks are invoked [48, 19]. The thread is then guided to its entry point function (the thread's start function) as specified by the developer. After the entry point function returns to the function which called it, *ntdll!RtlExitUserThread* is called, which terminates the thread [106]. Depending on the number of remaining threads in the process the process itself may be terminated if there are no more threads to be run.

A coarse outline of the functions called during the thread's life cycle is shown below:

1. *NtCreateThreadEx* or another function/mechanism starts the thread.
2. The thread starts execution at *ntdll!LdrInitializeThunk*. It is passed a pointer to a *CONTEXT* structure which holds both the address of the thread's entry point and the address of *ntdll!RtlUserThreadStart*.
3. *ntdll!NtContinue* is called with the aforementioned *CONTEXT* pointer as argument. *NtContinue* is a system call (see Section 2.1.9) which changes the processor context to the context structure given as argument. The destination *EIP* appears to always lead to *ntdll!UserThreadStart*.
4. *ntdll!RtlUserThreadStart*.
5. *kernel32!BaseThreadInitThunk*.
6. Thread entry point (returns to *kernel32!BaseThreadInitThunk*).

7. *ntdll!RtlExitUserThread*.

8. *ntdll!NtTerminateThread*. This is another system call.

System Allocations

When a process is created, Windows allocates a few special memory regions in the new process's address space that are unknown to the developer. In these allocations Windows keeps metadata about the process, its threads and other undocumented data such as locale data. The data can be accessed through the WinAPI, and the implementation details are meant to remain hidden to allow Windows to later change the structure of this data and their location. Most likely the reason some data is kept in user space is due to speed; the WinAPI can simply extract the data directly instead of having to do a context switch into the kernel. Because of this, only data that is not security critical is kept in user space this way.

Thread Environment Block The *Thread Environment Block (TEB)* is a 64-bit data structure which contains information about a thread. Each thread will have its own TEB. The location of the first allocated TEB seems to vary between 0x7EFD8000 and 0x7EFDB000, and every subsequent TEB created afterwards will be allocated at the location of the previous TEB minus 0x3000. The size of 0x3000 comprises one 64-bit TEB which spans two pages, and one 32-bit TEB which only requires one page. Both 32- and 64-bit TEBs are required since each thread has two stacks: one 32-bit stack and one 64-bit stack. Because the code is running in 32-bit mode despite the process being a 64-bit WOW64 process, the TEB being used is the 32-bit version.

To access the current thread's TEB one must use the WinAPI since its location is unknown. By reverse engineering the WinAPI we can, however, see that it is possible to access the TEB directly. Windows keeps a pointer to the currently running thread's TEB in the *fs* segment register, which is an outdated x86 register, nowadays used for other purposes. It is not possible to change this register in user mode, but it is possible to read the contents at which it points to by using indexing notation. Because we are not actually dereferencing the pointer ourselves, we do not know the address being pointed to. Luckily the TEB data structure has a member which is a linear pointer to itself, which can be read using its member's index (0x18).

Here are some examples of how the TEB can be accessed:

```
; Retrieve TEB.CurrentThreadId at TEB index 0x24.
mov eax, dword ptr fs:[0x24]

; Same as above, but by first retrieving linear pointer to self.
; This exact code is what happens inside the GetCurrentThreadId()
; WinAPI function.
mov eax, dword ptr fs:[0x18] ; NT_TIB* teb = TEB.Self
mov eax, dword ptr ds:[eax+0x24] ; teb->CurrentThreadId
```

Note that since this is an undocumented structure, an official structure definition does not exist, and variations may occur on different versions of Windows. The TEB's allocation does not follow the standard rule of being allocated at a multiple of 64kB (see Section 2.1.3), and it is not possible to deallocate or decommit within the allocation. It is however possible to modify the data. The TEB is an important data structure that we will revisit many times throughout this thesis.

Process Environment Block The *Process Environment Block (PEB)* is a 64-bit data structure which contains information about the process. It resides at location 0x7EFDE000 in user space, and its 32-bit counterpart resides at the first page adjacent to this page, at 0x7EFDF000. The PEB can be accessed through the TEB of any thread as the pointer member *ProcessEnvironmentBlock* (index 0x30). As with the TEB, for the most part only the 32-bit PEB is used, and is what will be referred to as the PEB throughout this paper. The PEB allocation which contains both the 64-bit and the 32-bit data structure can, as with the TEB allocation, not be deallocated or decommitted within, but the data can be modified.

The PEB can be accessed like this:

```
mov eax, dword ptr fs:[0x30] ; void* peb = TEB.ProcessEnvironmentBlock
```

SharedUserData A very special memory region which is in fact the topmost allocation in the address space (with exception of another single inaccessible block above it) is the allocation housing the *SharedUserData (KUSER_SHARED_DATA)* structure. This is yet another undocumented structure which contains shared data among all processes. The kernel updates this memory region very often and it is therefore a highly volatile region. Like the PEB and the TEBs it is

not possible to deallocate this region nor decommit inside it. Additionally, this region is only readable (it has the permission *PAGE_READONLY*, see Section 2.1.3), so it can in fact not be modified at all.

2.1.2 Handles

A *handle* is in simple terms an identifier which represents an object within the Windows kernel. It is an opaque data structure, typically word sized, which is meant to act as an abstraction to make working with Windows objects simpler.

For instance, instead of issuing a process's ID when requesting changes to the process, a handle should be used instead. This is because the process ID is public and every process may know it. On the other hand, only certain processes may possess the authorization to act upon said process, and therefore using only the identifier would be insufficient for security purposes. Instead, a process requests a handle to the target process, file, socket or other object, and if the calling process's privileges are high enough, the request is granted and a handle is returned by the operating system. The handle can then be used as an argument to WinAPI functions that accept the handle type (that is to say a file handle would not work as a process handle, and so on).

Another reason is reference counting. It is important for the operating system to know how many processes or threads hold a reference to an internal object so that it is not deleted prematurely. By forcing the use of handles the operating system has an interface in which it can keep track of which objects are being used by whom. When a handle has served its purpose it is therefore imperative that it is closed, otherwise the object which it referenced will never be deleted and a handle leak exists.

Different handle types may require the handle to be closed by the opposite function of which the handle was created. E.g., if a process handle was retrieved from the *OpenProcess* function, there is a chance there exists a corresponding *CloseProcess*. If this is not the case the generic *CloseHandle* function is used instead.

2.1.3 Virtual Memory

Allocations and Regions

Windows uses virtual memory which means that every process has access to the same memory layout. For a 32-bit address space this is 2 GB of virtual memory, and a memory access in one process will have no effect on the same memory address in any other process. The virtual memory is divided into pages of 4 kB each (not counting Large Page-support), which totals 524288 pages in a 2 GB address space. This is referred to as the *page granularity* of the system.

When memory is requested in Windows, the allocated block will always be aligned on a 64 kB boundary. This is referred to as the *allocation granularity* of the system. System allocations such as the TEB is an exception to this rule, and is aligned on page boundary. In a 2 GB address space there can therefore in total be 32768 allocations. Once an allocation has been created, it is not possible to resize it without freeing and reallocating it with a different size.

Memory allocation happens through the classic *malloc* and *new* (C++) functions, which in turn calls upon the heap manager. The heap manager calls *VirtualAlloc* when the pre-allocated heap space has been exceeded, which in turn calls *NtAllocateVirtualMemory* (which is a system call). The system call is bound by the granularity rules previously mentioned, and every memory allocation call eventually ends up at this system call.

An allocation can be thought of as a container for *memory regions*. A region is a set of pages grouped together which share some attributes. If two regions with different attributes are adjacent to each other and one of the regions changes its attributes to match the other, the regions will automatically merge into one big region. The distinction between an allocation and a region is very important to make note of. An allocation can contain one or multiple regions, but never zero.

Region Attributes

If a memory address within the address space is queried to retrieve information about the region in which it resides, Windows will return a data structure called *MEMORY_BASIC_INFORMATION*. The fields in this structure are as follows:

- **BaseAddress**: The starting address or the base address of the region. This address is aligned to page granularity.
- **AllocationBase**: The starting address of the allocation in which the region resides. For a new allocation, both *BaseAddress* and *AllocationBase* will be identical. This address is aligned to allocation granularity (with noted exceptions).
- **AllocationProtect**: The original protection of the region when the allocation was created. Region protection will be discussed shortly in Section 2.1.3.
- **RegionSize**: Self-explanatory. Aligned to page granularity.
- **State**: The state of the pages in the region (i.e. the region's state). The region can either be:
  - *Free*: The pages within the region are available for allocation.
  - *Committed*: Physical memory has been allocated for the pages within the region. To commit a range of pages within a reserved region, *VirtualAlloc* is used, not *VirtualProtect*, which only specifies the region protection. It is important to differentiate the two.
  - *Reserved*: No physical memory has been allocated, but the area is not free and can not be allocated. A typical example of a reserved region would be an array that has not yet been filled. If another allocation was made that stole part of the array due to the pages not being in use yet, that would create a gap and the memory would no longer be contiguous.
- **Protect**: The current protection of the region. Region protection will be discussed shortly in Section 2.1.3.
- **Type**: Defines the type of the region. A region can either be:
  - *Private*: The pages within the region are private and not shared with other processes.
  - *Mapped*: The pages within the region are mapped into the view of a section. Mapped sections are described shortly in Section 2.1.3.

– *Image*: The pages within the region are mapped into the view of an image section. Image sections are described shortly in Section 2.1.3.

Some of these attributes need further explaining.

Memory Protection

Memory protection refers to the page protection assigned to all pages within a memory region. Basically this refers to whether the pages are readable and writable. Two adjacent regions with different protections will remain as two independent regions and not merge, despite sharing other attributes. If an action is taken on a page that is not in correspondence with the protection of the region the page is in, an exception will be raised by the operating system.

We will here list the most common protection constants:

- **PAGE_NOACCESS**: Page access is disabled. Cannot read from, write to, or execute the page.
- **PAGE_READONLY**: Page can only be read.
- **PAGE_READWRITE**: Page can be read and written, but not executed.
- **PAGE_WRITECOPY**: Page can be read and written. Page modifications activates the *Copy-on-Write* mechanism and reverts the protection to *PAGE_READWRITE* (*PAGE_EXECUTE_READWRITE* for *PAGE_EXECUTE_WRITECOPY*).
- **PAGE_EXECUTE_READWRITE**: Same as *PAGE_READWRITE* with the added permission to execute the page.
- **PAGE_EXECUTE_WRITECOPY**: Same as *PAGE_WRITECOPY* with the added permission to execute the page.

The Copy-on-Write mechanism will be described in Section 2.1.4.

Page Guards Another mechanism of particular interest is the *page guard mechanism*. The *PAGE_GUARD* protection constant can be added to the original protection to enable guarding of the page. When an access is made on the page, whether it be a read, write or execute, an exception is raised by the processor which is caught by the operating system which in turns raises a *STATUS_GUARD_PAGE_VIOLATION* exception. This is very useful when one wants to monitor accesses to certain memory regions, and is frequently used in debugging. Additionally, thread stacks, once full and overflowing when new data is pushed on the stack, will touch upon an adjacent guarded page which triggers an exception from the operating system. In this case, however, the operating system will swallow the exception and instead move the guarded page further down so that the stack can be expanded. This repeats until the reserved stack memory has been exceeded.

Shared Memory

The allocations made within a process's address space are either private to the process or shared with other processes. If the allocation is private it is not accessible to other processes with the exception of when being queried directly with the help of the operating system. If the allocation is shared, other processes can access the memory and cause changes which will be visible in real-time to the other processes sharing the same section of memory. Shared memory should not be confused with memory shared between threads. All threads in a process share its address space.

A section of memory that is shared is called a *section object*. A section object is established by calling the *CreateFileMapping* function in the WinAPI. The developer decides whether the section represents a file on disk, or if a region within the system's *page file* should temporarily act as a shared area. The developer further decides what kind of protection should be applied to the section. Finally, the section is given a name that is global on the machine and it is specified how much of the file is being shared. For example, it is possible to specify the starting offset and size of the shared region.

Other processes and the process which established the section can then create a *view* into the section by calling *MapViewOfFile*. The function requires a handle to the section object it is supposed to view, and this handle can be obtained either from *CreateFileMapping* if the process itself created the section, or *OpenFileMapping*. The section name is unique across the machine so that other processes may open it through the use of this function.

Once a handle has been attained, the developer can specify how much of the section should be visible through the view, i.e. the start offset and size. The desired access must also be specified. If the required access is higher than that of the section object, the request is denied. A sister function of *MapViewOfFile* is *MapViewOfFileEx* which also allows the developer to decide at which base address the view should be loaded, otherwise the function will simply select an arbitrary base address at which it will place the view.

Once the view has been created, the memory can be accessed just as if it was a private region. Changes to the memory will not go through the operating system, because the caller has already gone through the authorization process, and all interaction with the shared memory happens in real time and is immediately visible to other processes with which the section is shared.

Last to mention is the *Image* memory region type. This is a special kind of a section object which maps to an executable file. If a view to an image is queried, the type of the region will be labeled as being an image. To create an image section, *CreateFileMapping* must supply the *SEC_IMAGE* argument for the section's protection attribute.

2.1.4 Libraries

In Windows, developers may choose to statically or dynamically add code libraries to their programs. Libraries that are static are included at compile time and cannot be swapped out later. Dynamic libraries on the other hand (in Windows known as *Dynamically-Linked Libraries (DLL)*), are independent files from the program executable and can only be interacted with at runtime. We will only talk about DLLs in this section. DLLs may be static or dynamic; they are either loaded during program start (static), or by using the *LoadLibrary* WinAPI function (dynamic). Windows system libraries that are included using the *#include* directive, for example, will be loaded statically.

System Libraries

In Windows, the most commonly used DLL system libraries are:

- **ntdll.dll**: The native interface to the operating system. Also known as *The Native API*. Most system calls are wrapped and issued by ntdll.dll. This library remains largely undocumented.

- **kernel32.dll**: Contains base functionality of the WinAPI such as memory management, I/O operations, process and thread handling, and synchronization operations.
- **user32.dll**: Contains functions to manage the user interface.
- **gdi32.dll**: Contains functions to perform primitive drawing for outputs such as video displays and printers.
- **msvcp*.dll and msvcr*.dll**: For programs compiled with Visual C++ the C standard library can be accessed through the msvc**.dll libraries. The asterisk indicates the version number of the runtime.

By keeping the system libraries as DLLs instead of static libraries, older Windows programs may still have a chance to function in future versions of Windows, as the DLLs may be updated independently from the program without the program's knowledge. A good example of this is programs compiled for 32-bit Windows. If static libraries had been used instead of dynamic libraries, it would not be possible to implement the WOW64 layer and support 32-bit programs on 64-bit Windows (WOW64 is described in Section 2.1.5).

DLL Events

Developers may specify their own entry-point function to a DLL. This entry point, often simply called *DllMain*, will receive notifications when certain system events occur. It is very dangerous to perform heavy work and use certain WinAPI functions from within this entry-point, because the program may deadlock. The reasons for this are outside the scope of this thesis, but are described in [70, 11].

The available events that the DLL may be notified about are:

- **DLL_PROCESS_ATTACH**: The DLL is being loaded into the address space of the current process either as a result of the process being started, or as a result of a call to *LoadLibrary*.
- **DLL_PROCESS_DETACH**: The opposite of *DLL_PROCESS_ATTACH*.
- **DLL_THREAD_ATTACH**: A new thread has been created (not issued on DLL load, only for threads created after the DLL has been

loaded). The event is caused by the new thread itself, and as such it is easy to retrieve information about the thread. This event is not raised if *DLL_PROCESS_ATTACH* was issued instead.

- **DLL_THREAD_DETACH**: The opposite of *DLL_THREAD_ATTACH*. The event is raised by the exiting thread.

Copy-on-Write

To save memory, multiple processes will share the memory regions containing the system libraries. This is made possible by using section views. Similarly, every process instance of a program will share the program's code and data. For this to work every program executable or library must have their memory regions marked as being an image, or a mapped region. Additionally, the regions must be marked as *PAGE_WRITECOPY* or *PAGE_EXECUTE_WRITECOPY*. When a copy-on-write region is written to, the operating system will stop sharing the region and give the process its own private copy, and the copy-on-write protection attribute will be removed. When this happens, more memory will be consumed due to the copied region taking up space, but the alternative would allow any process to tamper with any other process by modifying shared system libraries. The copy-on-write mechanism is not limited to program- or library images, and may be used with any shared section view.

ASLR

In older versions of Windows every system library always used to be loaded at the same address. This posed a security problem, because malware that managed to run code on the system could use the library functions to further exploit the system. This was possible because the malicious code knew the exact location of where the libraries were situated. This vulnerability also extended itself to the locations of the stacks, heap, PEB and TEBs.

To solve this class of problems *Address Space Layout Randomization (ASLR)* was introduced for Linux. It was later added to Windows Vista. The function of ASLR is to randomly arrange critical components of the address space at unknown locations. In Windows this happens at system startup and the chosen locations remain the same until the next reboot [50]. Programs and libraries must be explicitly linked with ASLR enabled, and

the option to disable ASLR both for the individual program and globally for all programs remains for compatibility reasons [108].

Some system libraries, notably *ntdll.dll*, *kernel32.dll* and *user32.dll* are required to remain in the same location in every process as the operating system will call into them from kernel mode (e.g., when creating a new thread *ntdll!LdrInitializeThunk* is the target callback). When the system boots the kernel will cache the locations of these system libraries until the next reboot [50].

2.1.5 WOW64

It is not possible to run 32-bit programs on 64-bit Windows because the system environment is different to that which is expected by the 32-bit program. For example, the system libraries will all be 64-bit, whereas the program is expecting a 32-bit library interface. To solve this problem and allow 64-bit Windows to be backwards compatible with 32-bit programs dating back to before the 64-bit era, the *Windows 32-bit on Windows 64-bit (WOW64)* was introduced.

WOW64 is a user mode emulation layer which makes the 64-bit address space look like a 32-bit address space, doing so transparently without the program's knowledge. When a WOW64 process is started, the 32-bit system libraries are loaded into the address space in addition to the original 64-bit ntdll.dll and the 64-bit WOW64 emulation libraries [79, 61]. All the 64-bit libraries remain hidden to the program. The memory can still be accessed directly (as with everything located in user space), but the DLLs have not been included in the PEB's *module list*, which is a linked list that contains all the loaded DLLs of the program.

The WOW64 DLLs that are loaded at program start are:

- **Wow64.dll**: Contains core emulation infrastructure and thunks for the Ntoskrnl.exe entry-point functions.
- **Wow64Win.dll**: Contains thunks for the GUI entry-point functions in Win32k.sys.
- **Wow64Cpu.dll**: Architecture-specific interface to switch the processor from 32-bit to 64-bit and back in addition to some thread context handling.

- **IA32Exec.bin**: x86 software emulator for Intel Itanium.
- **Wowia32x.dll**: Interface between IA32Exec.bin and WOW64.

System Calls

When the program interacts with the WinAPI, eventually the system library is going to perform a system call. In a WOW64 process the system call will instead be redirected to the WOW64 interface, which converts the 32-bit data structures to 64-bit and issues the system call in 64-bit mode. The operating system kernel is 64-bit and expects to be called from a 64-bit program. When the system call returns from the kernel, execution continues within WOW64 and the return values are converted back to 32-bit. This all happens transparently to the 32-bit program. Similarly, whenever the kernel needs to reach into user space, it will first pass through the WOW64 interface (more about such callbacks are described in Section 2.1.7).

The switch from 32-bit mode to 64-bit mode happens as follows. In kernel mode there exists a special data structure known as the *Global Descriptor Table (GDT)*. Within this table the operating system inserts *segment descriptors* which describes the various segments of virtual memory. The GDT is out of scope for this paper, but it is worth noting that there are two different segment descriptors set up for 32-bit and 64-bit execution. The difference between the two segments is that the 64-bit version has the *L-bit (Long Mode)* set, which indicates 64-bit execution [40]. Code running with the 32-bit segment descriptor active will run in compatibility mode (32-bit execution), and code running with the 64-bit segment descriptor active executes 64-bit code. The currently active segment descriptor can be changed in user mode by using a *far jump*, which is a long jump with a special prefix, but the contents of the segment descriptors may still only be changed in kernel mode.

In Windows 7 64-bit the two code segment descriptor indexes into the GDT are 0x23 for 32-bit mode and 0x33 for 64-bit mode (only a bit flip difference). The far jump is issued like this:

```
; Change code segment to 64-bit and jump to given offset.
ljmp 0x33:<offset>

; Another way to do the same transition to 64-bit.
unsigned long long jmpAddr = 0;
lea ebx, jmpAddr
```

```
mov dword ptr [ebx], <offset>
mov dword ptr [ebx+4], 0x33 ; code segment
jmp fword ptr [ebx] ; far word
```

This section described the details of how system calls are performed under WOW64. In Section 2.1.9 we will continue the discussion from a more general standpoint.

Threads

Threads running in a WOW64 process will have a 64-bit 256-512 kB stack allocated to them, which means that a 32-bit program that originally had a large number of threads may not necessarily be able to do so any longer, because the additional thread stacks fill the address space quicker [74, 94]. This should only be a problem in rare circumstances.

While the thread is running within the WOW64 emulation layer, the processor will be running in 64-bit mode, and therefore the thread will use its 64-bit TEB and 64-bit stack.

Other Changes Under WOW64

To achieve full transparency, WOW64 additionally redirects and handles [61]:

- Exception dispatching. Exception handling is explained in Section 2.1.8.
- Asynchronous Procedure Call dispatching (see Section 2.1.6) and User Mode Callback dispatching (see Section 2.1.7)
- File system redirection.
- Registry redirection.

For the most part the only changes made by WOW64 is to ensure that the data structures are properly converted between 32-bit and 64-bit.

2.1.6 Asynchronous Procedure Calls

When a thread is waiting for an operation to complete (e.g., I/O), it has the option to either spin or yield the rest of its time slice to let other threads

run. To let threads be more efficient Windows allows other threads (or the thread itself) to queue up function callbacks for the thread so that when it is not doing anything else it can execute the callbacks. These kind of callbacks are known as *asynchronous procedure calls* (APC), and every thread has its own APC queue. APCs are asynchronous because they are not executed right away when they are queued. The thread must first signal that it is in a *alertable state*. A thread enters an alterable state by calling *SleepEx*, *WaitForSingleObjectEx*, *WaitForMultipleObjectsEx*, *SignalObjectAndWait* or *MsgWaitForMultipleObjecteEx* with the appropriate arguments. When the thread enters an alertable state it will execute all the queued APCs before returning. This means that if the operation which the thread was originally waiting for completes, the thread will not handle it before the APCs have finished executing.

APCs are queued through the *QueueUserAPC* WinAPI function, but some other functions use APCs internally such as *SetWaitableTimer*. The kernel may also queue APCs to be run in user mode. An example of this is when a new thread is created, *ntdll!LdrInitializeThunk* will be queued as a user APC (see Section 2.1.1 for more about the life cycle of a thread) [49, 89]. There exist two types of APCs for use in the kernel and one for use in user mode, but we are only concerned about the *user APC* type when we refer to APCs.

Dispatching

When a thread enters an alertable state the kernel will iterate over all the callbacks in the APC queue and call them sequentially. The kernel calls into user mode at the *ntdll!KiUserApcDispatcher* function which takes as arguments the callback to be run, the context snapshot of the system call entering the alertable state, and three arguments which will be passed to the callback. The function uses a custom calling convention and the stack will look as follows when entering the function:

```
ESP+00: ApcRoutine (function pointer)
ESP+04: ApcArg1
ESP+08: ApcArg2
ESP+0C: ApcArg3
ESP+10: Context (not pointer, full CONTEXT structure)
...
```

The dispatcher invokes the APC callback and passes it the three arguments which was passed to *ntdll!KiUserApcDispatcher*. These are the arguments that were given to the function which initially queued the APC. Before issuing the call the dispatcher wraps it in a SEH try-catch block in case the callback raises an exception (see Section 2.1.8 about exception handling). After the callback has completed the dispatcher calls *NtContinue* with the context argument passed to *ntdll!KiUserApcDispatcher*. *NtContinue* is a special system call which changes the processor context and transfers execution to that which is stated by the *CONTEXT* structure (a Windows data structure). The dispatcher therefore redirects the execution back to the system call which initially caused the thread to enter an alertable state.

It is worth noting that when *ntdll!KiUserApcDispatcher* is entered the stack will be at a lower point than before the dispatcher was called, as additional temporary data resides on the stack.

Dispatching Different Callback Types

The standard dispatching process which was just described has a caveat. When WinAPI functions queue APCs, they may use up to three arguments, as shown above. These arguments are passed to the callback and are accessible as normal function arguments within the callback. However, some WinAPI functions, e.g., *QueueUserAPC*, only allows the developer to pass in a single argument. The calling convention remains the same and therefore all three arguments will be passed in, even if they are not being used.

For single-argument APCs like this, the dispatcher works a little differently. The *ApcRoutine* argument on the stack will point to *ntdll!Ordinal8* (also known as *kernel32!BaseDispatchAPC*), which is an initialization routine that converts the three-argument APC into the expected single-argument APC [99, 45]. Some additional initialization and clean up is performed as well. The actual APC callback will be put in *ApcArg1* on the stack, and its single argument in *ApcArg2*. As a result, the dispatcher will call *ntdll!Ordinal8*, believing it is the actual APC callback. Before *ntdll!Ordinal8* calls the actual APC callback, it first copies the single argument to the appropriate location on the top of the stack and then calls the callback. The argument can then be accessed normally as if the callback only accepted a single argument.

2.1.7 User Mode Callbacks

The graphics sub-system in Windows used to create GUIs is implemented in two places: *user32.dll* and *win32k.sys*. The latter is a driver written by Microsoft, containing a large amount of functions. Certain window management tasks that are best handled in user mode need to be run from within the driver, and this is done through *kernel-to-user callbacks* or *user mode callbacks*. This mechanism is intended for internal use only and is not publicly documented. Many of the functions in user32.dll act as callbacks to this mechanism. We note that user mode callbacks always happen as the result of a user mode system call into the kernel first [15, 47].

Transitioning to User Mode

User mode callbacks can be thought of as similar to a reverse system call. The kernel will transfer control over to *ntdll!KiUserCallbackDispatcher* with an *ApiNumber* index, a pointer to a buffer, and the length of said buffer [29, 51]. The ApiNumber is used to index into an array held in the PEB which contains function pointers (callbacks), all of them located in *user32.dll*. The kernel ensures that the user mode stack has enough space for the buffer whose size depends on the specific callback. The input buffer is then copied from kernel space to the user stack. It contains the arguments to the callback which are packed in a custom structure. Before the callback is invoked the dispatcher wraps it in a SEH handler (see Section 2.1.8) to secure against unhandled exceptions, and finally the callback is executed [15, 47].

Similar to how APCs are dispatched, the *ntdll!KiUserCallbackDispatcher* also uses a custom calling convention where the arguments are placed on the top of the stack before the transition is made into user mode. We also note that the stack pointer remains at a lower address than when the system call causing the user mode callback was run, as temporary data resides on the stack to be used by the dispatcher.

Returning to Kernel Mode

After the user mode callback has finished it must return to the kernel. There are two ways to do this. The first is to simply return from *ntdll!KiUserCallbackDispatcher* which will issue a *NtCallbackReturn* system call and use default arguments. The second is to issue *NtCallbackReturn* directly and include a pointer to an output buffer and its size. By issuing the

system call directly, the execution does not return to *ntdll!KiUserCallbackDispatcher* [15, 47].

2.1.8 Exception Handling

For exception handling Windows uses its own basic mechanism which others (e.g., C++ compilers) build upon. The Windows exception handling mechanism is known as *Structured Exception Handling (SEH)*. SEH is a large topic, therefore we will only describe the concepts that are directly relevant to this thesis. The following explanation of SEH is a summarized version of the description found in [85].

The Life Cycle of an Exception

When an exception occurs, Windows will gather all the necessary information about the exception and call a handler function (an exception handler). The handler is given the information about the exception and uses this information to do what it needs to do. It then returns a value to the operating system indicating what to do next.

Because every thread may be within its own nested hierarchy of exception handlers (try-catch blocks), Windows needs to keep track of the exception handlers on a per-thread basis. The first member at index zero of the TEB contains a pointer to the thread's most recent *exception record*, a data structure which holds an exception handler callback and a pointer to the previous exception record one level up in the hierarchy. Combined, these exception records form a linked list.

Windows will follow the pointer found in the TEB to the first exception record for the faulting thread, which is the most recently installed record. The handler found in the record will be executed and its return value examined. The return value decides whether the faulting instruction should be re-run or whether the SEH mechanism should continue to look for another handler further up in the hierarchy that may be better suited to handle the exception. The operating system would in that case follow the current exception record's member which points to the previous record.

If no handlers are found that can handle the exception, Windows ends up at the top-most handler which is the *unhandled exception handler*. This handler was installed by the operating system before the thread's entry point was run.

Adding and Removing Exception Handlers

The aforementioned linked list of exception records are purposely placed on the thread's stack so that when the function in which the exception handler resides returns it is an easy job to also get rid of the exception handler. To create an exception record and add it to the front of the linked list, this is the traditional way to do it:

```
; Start creating an EXCEPTION_RECORD structure on the stack:
; Record.Handler = <exception handler function pointer>
push <exception handler function pointer>

; Link the new record to the previous record:
; Record.PrevRecord = TEB.CurrentRecord
push fs:[0]

; Add the new record to the front of the list:
; TEB.CurrentRecord = Record
mov fs:[0], esp
```

The exception record can be removed with the following code:

```
; Assumes stack has been popped to the point
; where the exception record resides:
mov eax, [esp]

; Install previous record:
; TEB.CurrentRecord = TEB.CurrentRecord.PrevRecord
mov fs:[0], eax

; Clean the exception record structure off the stack.
add esp, 8
```

These series of steps is essentially what C++ compilers for Windows do under the hood when compiling try-catch blocks. In reality the creation and handling of exception records is much more complicated as the data structure has been extended in newer versions of Windows, but as mentioned previously the discussion here should suffice for our needs despite this, as will be evident later when we describe how we record exceptions (Section 5.3.2).

Exception Dispatching

When the processor raises an exception or a software exception is raised by the program, the kernel will catch the exception and transfer control to *ntdll!KiUserExceptionDispatcher*, assuming the process is not being debugged, in which case the debugger will be notified of the exception and given control over the execution first. This is another callback very similar to the aforementioned *ntdll!KiUserApcDispatcher* (Section 2.1.6) and *ntdll!KiUserCallbackDispatcher* (Section 2.1.7). The exception dispatcher callback receives two arguments: a pointer to a context structure and a pointer to an exception record. The context argument contains the execution context of the thread as it was before the exception occurred, and the exception record argument points to the first exception record in the linked list of exception records.

The dispatcher figures out which handler should be called and the exception is processed according to the description presented earlier in this section. *Vectored Exception Handlers (VEH)* will also be processed within the dispatcher if they exist, but they are out of the scope of this thesis. If a handler successfully handled the exception, the context structure argument is realized by a call to *NtContinue* and execution continues at the instruction which caused the exception so it can be re-executed. If no handler was found, the dispatcher will raise another exception which can not be caught and handled (it is marked as *non-continuable*), and the kernel will notify the debugger a second time (if any) or otherwise exit the program [85, 44].

As with the previously seen callbacks into user mode the arguments to *ntdll!KiUserExceptionDispatcher* are delivered on the stack using a custom calling convention. The stack is therefore being used to harbor the temporary data used by the dispatcher.

KiRaiseUserExceptionDispatcher

Exceptions that are intentionally raised through the *NtRaiseException* system call also end up in *ntdll!KiUserExceptionDispatcher* despite there existing another dispatcher with a name that more closely resembles the system call name: *ntdll!KiRaiseUserExceptionDispatcher* [46]. This dispatcher simply creates an exception record and raises a new exception from the record with another call to *NtRaiseException*. The dispatcher is rarely used in practice and is therefore not of much interest, but we would still like to record it,

therefore we mentioned it at this stage. The dispatcher uses a normal calling convention and takes no arguments.

2.1.9 System Calls

System Call Types

In early versions of Windows system calls were issued by using the *int 0x2E* instruction. The interrupt handler at index 0x2E was run and the system call handler corresponding to the given system call index executed thereafter. In machines with a x86 Pentium II processor or newer the *sysenter* instruction is used instead, which is designed to be faster than the traditional way. The system call index is placed in the EAX register and a pointer to the system call's arguments placed in EDX before the system call is issued. This is true for both the old and the new method. Despite there existing a faster way to perform a system call the interrupt method is still supported for compatibility reasons. On the x64 architecture Windows instead uses the *syscall* instruction. The system call index is passed in the EAX register, the first four arguments in other registers, and any remaining arguments are passed on the stack [61]. Because WOW64 processes have their system calls redirected to the WOW64 layer (see Section 2.1.5) all 32-bit programs being emulated under WOW64 also use the *syscall* instruction instead of *sysenter*.

Issuing System Calls

Programs never issue system calls directly (with the exception of very rare circumstances and malware) and always use the native API which wraps system calls (see Section 2.1.4). The standard template for a system call invocation in a 64-bit process looks like this:

```
mov r10, rcx
mov eax, <system call index>
syscall
ret
```

In a WOW64 process the procedure looks a little different:

```
mov eax, <system call index>
xor ecx, ecx
lea edx, dword ptr ss:[esp+4]
```

```
call dword ptr fs:[0xC0]
add esp, 4
ret <variable size>
```

As we can see the EAX register is still being used to contain the system call index, but in this case the EDX register is also used to hold a pointer to the system call's arguments. This is because 32-bit programs would normally use the *sysenter* instruction, in which this is the calling convention. It is the responsibility of the WOW64 layer to convert these arguments to what is expected by the *syscall* instruction's calling convention. The *call* instruction is what actually transfers the control over to the WOW64 layer, as the target of the call is the aforementioned far jump which transitions the processor into 64-bit mode. The address of the far jump instruction is fixed and only changes after a reboot. It can be found in the WOW64 TEB at index 0xC0, which is why the *fs* segment register is being dereferenced.

WOW64 System Call Calling Convention

The system call interface follows a calling convention as mentioned previously. The calling convention for programs executing with WOW64 emulation is listed below. It is important to note that from the perspective of the 32-bit program the system call interface resides within the native API – not the WOW64 layer. 32-bit execution can not continue past the far jump which leads to WOW64, therefore from the perspective of the program this far jump is the system call interface. In other words, the system call interface will be using a 32-bit calling convention despite the fact that the WOW64 layer performs a 64-bit system call and uses a 64-bit calling convention.

The system call interface calling convention for WOW64 is as follows:

- The system call index is placed in EAX. A numeric argument is placed in ECX (or is NULL). A pointer to the system call arguments is placed in EDX. The system call's return value is placed in EAX (this will be an NTSTATUS return code). In essence, Windows system calls use the *stdcall* calling convention in which all but the EAX, ECX and EDX registers are preserved [28].
- EFLAGS is not be preserved.
- FPU (x87) and MMX registers are preserved [73, 78, 28]. The 64-bit

kernel will not use these register extensions either way due to their deprecated status [78].

- SSE registers are preserved [78, 77, 72].
- AVX registers are preserved [77].

2.2 Program Analysis Background

In this section we will describe common techniques used to perform program analysis. This section should be thought of more as a utility section, as the techniques are not always necessarily directly intended to be used for program analysis, but reside in the corner between multiple categories such as reverse engineering, dynamic program analysis and program manipulation.

2.2.1 Hooking

A *hook* is a mechanism that redirects a program's execution flow. In essence it is either a *jmp* or a *call* instruction that overwrites the original bytes at a specified location in the program. When the execution reaches the hook, the program will be redirected to the hook's target instead of executing the original bytes.

Many Windows API functions famously include filler instructions in their function prologue that essentially do nothing (e.g., *mov edi, edi*) so that a hook can be installed if necessary without overwriting the actual function prologue and thus risking that a thread may already be executing at the location where the hook is being installed (this concept is known as *hot-patching*) [13]. For most other hooks however this luxury cannot be afforded and the original bytes must be backed up if they are to later be restored. It is most common to hook the beginning of a function or block of code, but hooks can be installed on any instruction as long as the remaining instructions surrounding the hook are left in a valid state. That is to say that if a two-byte instruction had a hook installed that overlapped on its second byte, the instruction would no longer be valid as only its first byte would remain. It is likely that the two-byte instruction would turn into an unrelated single-byte instruction and crash the program or corrupt its data.

Installation

When installing a hook, it is imperative that the installation happens in an atomic fashion. If a thread executes the hook before all of its bytes have been installed (thereby leaving an unfinished instruction) disaster will ensue. For this reason all the threads that may happen upon executing the hook must be suspended while the hook is being installed.

A better alternative is to install the hook by using an atomic instruction which copies all the bytes at once. Atomic instructions only work with sizes that are a power of two (2, 4, 8, 16, etc.), therefore for a five byte hook (*long jmp* or *call*) the nearest size is 8. This leaves three bytes which must be copied from the original bytes and then used as padding. An example of an atomic instruction that can copy eight bytes at a time is *cmpxchg8b (used with lock prefix)* [60].

It is also safer to hook API functions at the highest possible abstraction level; e.g., instead of hooking the system call issued by a function, hook the function's public interface instead. This is especially important when hooking the WinAPI, because locks may be acquired by the function before the system call is reached. If the system call is redirected to the hook's destination and there happens to be another WinAPI call there that attempts to acquire the same lock we will have a deadlock.

Trampolines

A hook is a one-way redirection mechanism and will not return to the original code. If the code at the hook's destination location intends to return to where the execution was redirected from the original bytes must first be executed to preserve the original execution flow of the program. Neglecting to do this will most likely crash the program.

An extended variant of the classic hooking mechanism is what is called a *trampoline*. A trampoline works like a normal hook but will also return back once the code at the hook's destination location has finished executing. Installing a trampoline is therefore not as easy as overwriting a few bytes. The trampoline must reserve a block of memory large enough to hold the original bytes overwritten by the hook, another hook that is used to return from the trampoline, and some additional instructions that invoke the hook's location code while preserving the processor's context. It is important that the code performing the transitioning does not unintentionally change any-

thing as the transitions are taking place, because the hooking mechanism must appear transparent to the execution.

2.2.2 Dynamic Binary Instrumentation

Code instrumentation is the act of modifying pre-existing code to allow for program analysis. *Static instrumentation* can be added at compile time, but is inflexible when it comes to analysing pre-compiled programs. *Dynamic instrumentation* solves this problem by instrumenting the program as it executes. We will in this section introduce the concept of *dynamic binary instrumentation (DBI)*, which as the name implies is dynamic instrumentation targeted at native programs.

Dynamic instrumentation can be very helpful in automatically monitoring a program's execution to find errors. It is common to use an already existing DBI framework due to the inherent complexity in implementing one. Some of the most well-known DBI frameworks are *Pin* [1], *DynamoRIO* [2] and *Valgrind* [3]. Tasks such as finding memory leaks, uninitialized variables, data races or program intrusions can be made easier by using dynamic instrumentation [80]. The frameworks are often generic and allow developers to write "tools" (referred to as *clients*) that are given access to callbacks at certain instrumentation points during the execution, for example on every instruction or every memory access, or at events such as exceptions or thread starts and terminations.

Implementation

Typically, a DBI framework will inject itself into address space of the process it wishes to monitor (e.g., by using DLL injection, ref. Section 2.2.4), or otherwise gain a foothold on the program execution. The original program instructions are read and analyzed in blocks and instrumentation points are added wherever needed as specified by the DBI client. It is normal to group and analyze instructions until a flow-control instruction such as a *jmp* or *call* is encountered, thereby ending the group. Such a group, or block, is referred to as a *basic block*. The process of analyzing and instrumenting basic blocks repeats throughout the program execution. Every instrumented

[1] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[2] http://www.dynamorio.org/
[3] http://valgrind.org/

block is cached, and if the execution path passes through the same basic blocks, as is often the case, there will be a significant performance boost.

Note that the original program instructions may not be executed at all; they are only read and a new instrumented block is created which is what will be executed in place of the original block.

Transparency

It is extremely important for a DBI framework to do its best to remain invisible to the program it is instrumenting, otherwise the program may behave differently and take a different execution path. For this reason DBI frameworks often employ complex techniques to remain transparent to system libraries, the stack, the heap, I/O, synchronization, floating point operations, thread operations, and so on.

This extends to DBI clients and DBI frameworks therefore typically encourage client writers to do all information retrieval through the framework's API. For instance, Pin overwrites some values in the TEB and if the tool relies on this information to make a decision it may make the decision based on incorrect information. By instead accessing the data through the *PIN_SafeCopy* function the client will receive the original data (because Pin keeps it stored somewhere else) [41].

Performance

DBI frameworks use complex caching policies in an attempt to avoid having to analyze and instrument basic blocks more than necessary. The cache store is often referred to as a *code cache*. Despite using code caches DBI frameworks suffer substantially when it comes to performance and it is not unheard of to receive a slowdown by a factor of 5-100 times [80, 57]. For instance, *Nulgrind* which is the most basic Valgrind client which does nothing has an overhead of 5x [39]. Depending on the target program the slowdown may be acceptable, but it is particularly unfavorable for interactive programs or programs that rely on strict timing such as network communication with a server.

2.2.3 Software Emulation

Emulation is the concept of reproducing the functionality and behavior of something else. In the context of computing emulation aims to reproduce the

behavior of code that is foreign to the host system, e.g., running ARM code on a x86 system. There are multiple ways to do this, the most common being software emulation and virtualization. In this section we will only discuss software emulation. An *emulator* is a component which performs emulation.

Implementation

As the name implies, software emulation is emulation performed in software. The emulator reads the foreign code and has been specifically programmed to recognize the target code's instruction set. It can therefore read the foreign code. To execute the instructions the emulator must be aware of the instruction set's specifications and emulate every instruction according to the specification to achieve the desired results. The most basic way to do this is to have a loop with a gigantic switch-statement which covers every instruction. The opcode of the instruction is read and passed to the switch-statement which then finds the appropriate handler for the instruction. The emulated registers and memory are updated according to the effects caused by the instruction. The loop then proceeds to read in the next instruction, and so on.

Because emulation attempts to reproduce the behavior of something else there is always a risk that the emulation is not completely faithful. Software instructions tend to have many side effects and it can be difficult to properly emulate all of them. For example, if an instruction has a bug in the real hardware, this bug must also be emulated. Implementing an emulator therefore comes with a high implementation cost.

Performance

Software emulation comes with a very obvious performance penalty. For every instruction being executed there are potentially hundreds of other instructions that must be run in order to emulate the instruction. The result may be a performance slowdown of a factor of several hundred, and it is therefore extremely important to have a good caching policy to avoid having to re-translate instructions all the time [4, 110]. On the other hand, the software emulation has total control over the instruction stream. It is possible to inspect every instruction before executing it, and also to examine the register and memory state at any time. Virtualization in comparison does not offer this level of control.

2.2.4 DLL Injection

It is possible for a process to load arbitrary DLLs inside the address space of another process. This technique is known as *DLL injection*. The technique can also be used to execute code remotely in another process with minor modifications. DLL injection can be used to intrude on a running program's execution by invading its address space and directing the execution to do things the authors of the program did not expect. After the DLL has been injected it is possible to invoke code which resides in the DLL image and therefore assume control over the program.

Assuming the calling process has the sufficient privileges to perform the access operations on the target process, the way to inject a DLL into another process goes as follows:

1. Use the *GetModuleHandle* WinAPI function to retrieve a handle to kernel32.dll. Then use the WinAPI function *GetProcAddress* to extract the address of the *LoadLibrary* WinAPI function from kernel32.dll.

2. Allocate memory in the target process by using *VirtualAllocEx*. The allocation must be big enough to fit the file path of the DLL to be injected, including the null-terminator. Next write the DLL file path to the allocated block by calling *WriteProcessMemory*. For this to work the allocation must be writable, so we ensure its protection permits writing. The DLL path now resides in the target process in the allocated block.

3. Create a remote thread in the target process. This can be done with a call to *CreateRemoteThread*, but it is important to block until the thread exits by calling *WaitForSingleObject* and passing in the thread's handle, otherwise the thread creation function will return instantaneously before we have completed our task.

   Every new thread takes an entry point function, and the function we specify as the thread's entry point is the address of the *LoadLibrary* function which we extracted earlier. Because kernel32.dll is shared by all processes through shared memory the function address will be the same in both processes. The thread entry point function can also receive an address, so we pass the address of the allocated block in the target process as the entry point function's argument.

The way *LoadLibrary* functions is that it takes a single string pointer argument which holds a DLL file path, and then loads the DLL file which is specified by the path. By passing in the address to the allocated block we are passing the *LoadLibrary* function the path to the DLL we want to be loaded. This address will be valid since it resides in the target process's address space.

4. The DLL has now been loaded into the target process's address space as if the process itself performed the *LoadLibrary* call.

5. We free the allocated block in the target process by calling *VirtualFreeEx*.

6. We query the thread's exit code by calling *GetExitCodeThread* and passing in the remote thread's handle. The value returned by the thread's entry point function will be the thread's exit code. In this case it is the return value of *LoadLibrary* which is the newly loaded DLL's base address within the address space.

Chapter 3

Overview of Deterministic Replay

The research area of *deterministic replay*, or *record and replay (R&R)* as it is also known as, has been studied for a long time both by academia and the industry [14]. The main focus of the research is the ability to *record* a program execution and later *replay* it with full fidelity, preferably indefinitely. In and itself this would not be particularly useful, but for programs that contain a significant deal of non-determinism – events that can not be predicted before running the program (e.g., user input, network communication, etc.) – the ability to re-examine the program without having to reproduce these events in the exact same order is tremendously helpful. Tasks such as intrusion detection, program analysis and especially debugging suddenly become trivially easy compared to traditional approaches. As an added bonus, because the recorded log is a physical file, it can be shared with others, parsed by scripts and analyzed automatically, even in parallel. Since each action taken by the original program has been immortalized in the recording, it is possible to execute in the backwards direction, opening up a whole new world of algorithms and approaches to analysis.

A tool that makes this all possible is known as a *deterministic replay tool*. In this chapter we present the problems solved by deterministic replay and some of its popular applications, the taxonomy of deterministic replay tools and important concepts central to the field.

3.1 Program Tracing

When developers want to understand what is happening inside a running program they often use techniques to trace the execution of the program [111]. Many tools, e.g., debuggers, offer this functionality built-in, but stand-alone tracing tools exist.

Depending on the tool and its configuration, execution logs may be sparse or abundant with detailed information. The more detailed the logs, the heavier the logging overhead in both space and time, but this comes with the trade-off that detailed logs are often more useful to the developer.

Additionally, the *implementation cost* of the tracing tool is affected by the level of detail being logged. For example, if the tracer only logs function calls, it may be enough to disassemble and hook anything that resembles a function, then redirect the hooked function prologues to the function which does the logging. However, if the tracer wants to log all the instructions executed, the previous technique will not be sufficient and the implementation cost goes up. Similarly, if the tracer wants to log all memory accesses as well, yet another technique must be employed that may be much more costly to implement compared to the other two.

3.2 Deterministic Replay

A *deterministic replay tool* is a more advanced form of a tracing tool, dedicated to assist in recording non-deterministic program executions. A deterministic replay tool should in general not be confused with a typical tracing tool, because such a tool tends to output plain log files and does not necessarily facilitate replaying the program's execution from those logs. Often, the logs contain data in an easily parsable format specific to the current use case, which lends them to quick analysis using scripting languages.

A deterministic replay tool on the other hand focuses on letting the developer interactively inspect non-deterministic phenomena in a deterministic manner. The program's execution log is formatted in such a way (not necessarily human readable and often compressed) to facilitate the ability to replay the full program execution indefinitely. All non-deterministic interaction with the process's address space is recorded so that it can be reproduced during replay. Once all non-determinism has been eliminated by recording the program run, the developer can analyse the execution in a stable envi-

ronment without having to resort to cyclic debugging.

3.3 Sources of Non-Determinism

In a single-threaded program on a single-processor system, there exist a multitude of non-deterministic factors that perturb the program's normal execution path from being entirely deterministic. We therefore need to record these events to be able to replay the program execution with high fidelity.

Because non-deterministic events occur relatively infrequently they can be recorded efficiently ([14], [104]). To accurately replay the recorded non-deterministic events, both the time and location at which they occurred must be logged in addition to their data [3]. However, depending on the level of abstraction being recorded at, not all events necessarily have to be recorded. For example, a user mode deterministic replay tool is not interested in recording every single hardware interrupt even though the operating system has to handle those. The recorder is only concerned with recording what has visible effects, so as long as the interrupts do not affect the recorded process's address space, they do not have to be recorded either [25].

We will here list the most common resolution to each of these non-deterministic events:

- **Non-deterministic instructions.** There exist a few instructions that are non-deterministic in nature. An example is the x86 *rdtsc* (ReaD TimeStamp Counter) instruction whose result is determined by the wall clock time at which it is executed. Other non-deterministic instructions act similarly and have the possibility to return variable results on each execution. The most common way to record non-deterministic instructions is to instrument the program binary and record their return values [14]. We will discuss non-deterministic instructions more in depth in Section 4.4.2, because they are particularly challenging to record in user mode.

- **Non-deterministic functions.** Some library functions found in user space can return non-deterministic results. For example, randomness functions, time or date functions, user input etc. The common way to record these functions is to add a wrapper around the function. This allows us to not only record the function's return value but also its side effects, if any [14]. The wrapper must also exist during replay so that

the wrapper can return the logged results from the record run. It is possible that the result returned from non-deterministic library functions come from a deterministic source at a lower level of abstraction. For instance, the operating system may use a deterministic counter variable that the library function returns the value of, but from the perspective of the library function's interface it appears as if the returned values are non-deterministic.

- **System calls.** Many system calls yield non-deterministic results, both in return values and other side effects. For instance, the system call to query the current process's ID (*getpid* on Linux) naturally will not return the same value for different processes (note that Windows does not use a system call for this purpose as the process ID is saved in the TEB). The common way to cope with recording system calls is to wrap the system calls and record their return values, then inject the result at the correct time during replay without actually executing the system call [14].

- **Signals and interrupts.** Interrupts are a way to allow the hardware and the operating system to communicate with the running program. Peripheral devices such as mouse and keyboard are prime examples of entities that produce interrupts and alert the operating system when they are being used so that the operating system in turn can alert the program. A well-known interrupt is the timer interrupt caused by the processor's internal clock to signal that the current thread's time slice is over and a new thread should be scheduled. The timer interrupt may very well be the most prominent source of non-determinism in this list (e.g., for a 3GHz processor with a 100KHz timer frequency the interrupt will occur every 30.000 instructions [14]). That is, when the timer strikes a new thread may be scheduled, and thread scheduling order has a strong effect on what happens during the program run (e.g., data races are based on the scheduling order).

  Signals are similar to interrupts in that they communicate with the program, but signals are a software construct (a form of inter-process communication), whereas interrupts are mainly caused by hardware. Signals allow other programs and the operating system to control the execution flow of a program (e.g., SIGKILL signals to terminate the program).

Both signals and interrupts may significantly affect the behavior of a program's execution. They are asynchronous in nature and may occur at any instruction, and are therefore difficult to record for some deterministic replay tools (particularly those implemented in user mode). The real timing of a signal or interrupt tend not to be recorded, instead the instruction at which it occurred is recorded with enough context to distinguish it from the other executions of the same instruction throughout the program execution.

When replaying signals or interrupts they are injected at the instruction at which they occurred, even though no signal or interrupt was processed by the operating system. Instead of originating from their original source (e.g., hardware), their information is pulled from the replay logs and reproduced.

A common way to record signals and interrupts is to modify the operating system, because all signals and interrupts are handled by the operating system at predefined locations before communicating with the respective program.

- **I/O operations.** Input/Output devices, such as the hard disk or network interface controller, may be manipulated by the program through mapped memory. Because the devices are likely to have new state when we want to replay at a later time, we can not trust those devices to act the same way as they did in the recording. This can be solved by recording all load and store operations, but only perform the load operations during replay by pulling the loaded values from the log and ignoring store operations [14].

- **DMA operations.** Direct Memory Access (DMA) is a special I/O operation which copies data from and to main memory from peripheral devices with the assistance of a coprocessor ([14], [80]). A DMA operation may occur concurrently at any time, in any direction and with any value, and is therefore difficult to predict and record, giving it an infamous reputation in the deterministic replay world. Dealing with DMA is out of scope for this paper, but interested readers can find more information on the three standard ways to solve it in [14].

3.4 Taxonomy of Deterministic Replay

A survey conducted in 2015 by Chen et al. [14] aimed to create a taxonomy of deterministic replay tools. As tracing tools are similar to deterministic replay tools, what was discussed previously in Section 3.1 about trace logs and overhead also applies to deterministic replay tools, and the survey helps classifying these observations into quantifiable criteria. We will in this section look at how to classify deterministic replay schemes into categories, but first we will discuss what are the general challenges of designing a deterministic replay scheme.

3.4.1 Technical Difficulties of Deterministic Replay

The five technical issues that are often seen in deterministic replay tools are log size, record slowdown, replay slowdown, implementation cost and probe effect [14]. Interestingly, each of the presented issues can be effectively addressed if the other issues are ignored. On the other hand, it is hard to simultaneously address multiple issues, as improvement on one issue might lead to exacerbation on another issue. This is especially true for multi-processor deterministic replay tools where shared memory between processors is the cause of significant non-determinism compared to single-processor systems. For such tools it is practically impossible to perfectly satisfy all five criteria because they can be considered as mutually exclusive. This is somewhat less problematic for single-processor deterministic replay tools, but there is a clear correlation between them [14].

Log Size

Even though disk space is cheap this criterion should be fairly obvious. As with most files, the smaller the size, the better. There are multiple ways to achieve this goal. First, one can use compression. Second, it is not necessary to store information unless it changed [14]. If the log size is considerably large, it becomes difficult to record long program runs, because the machine would eventually run out of disk space. This is particularly true for logging schemes that include a great deal of information. Performance would also be affected, as the logging operation would incur some overhead [14], this also being correlated to the amount of detail being logged. For a respectable deterministic replay tool this should not be an issue, however, because for

the most part it is only the non-deterministic events that need to be logged, and not everything in between.

An observation made by the author of BugNet [80] and the authors of iDNA [8] that can have a considerable impact on log size is that not all memory reads need to be recorded; only reads that cannot be predicted must be recorded. BugNet only records the result of each *first load*, that is, the first time a register or memory location is accessed after the most recent cache miss, interrupt, system call or checkpoint [14]. The remaining load operations can be inferred with the help of the recorded first loads. Similarly, iDNA compares read memory values against a cache and only logs the read value if the corresponding value in the cache did not match.

For a multi-processor deterministic replay tool on the other hand, log size can be a major problem due to the fact that the order of shared memory accesses need to be logged. A great deal of research is currently ongoing to improve this, and many solutions have already been suggested [14].

Record Slowdown

Record slowdown or recording overhead is a very important factor in how usable one can consider a deterministic replay tool to be. If the system is meant to be used interactively while being recorded, the reduced responsiveness of the system will be visible to the user. Additionally, if the system to be recorded has to stay online for days, weeks or worse, a considerable speed penalty will greatly increase the overall runtime of the system. An example is deterministic replay tools employed to help with continuous testing. For instance, rr [82] is used daily to debug Firefox, but to do that they first need to record tens of thousands of test runs. If those test runs were to be considerably slowed down, the whole debugging cycle would be slowed down ([82], [3]). Anderegg argues in his thesis [3] that performance is the most important characteristic to enable a deterministic replay tool to be used in practice.

According to the survey [14], recording overhead is especially an issue for deterministic replay tools implemented in software due to their lack of hardware support. Hardware-assisted schemes do not need to execute extra instructions to record the program, because dedicated hardware takes care of that job behind the scenes, hence it performs better (but may suffer from large log sizes due to the massive amount of memory and I/O transfers, which in itself can be a source of slowdown). On the other hand, schemes imple-

mented in software-only need to execute additional instructions to perform the logging step, and that can be expensive. This is especially problematic for multi-processor deterministic replay tools, because they need to record all shared-memory accesses (as an example, PinPlay [84], a software-only scheme, has more than 100x overhead because of this [14]).

Alternative ways have been proposed to ameliorate this, as mentioned in the survey [14]. For instance, instead of recording the shared-memory values, their orderings can be recorded, but that may require the program to be free of data races. Another example is to log less information, but then more time has to be taken in the replay execution to deduce the missing links by replaying over and over. Researchers have also attempted to make parallelism deterministic so that recording shared-memory can be avoided altogether, but this introduces a larger probe effect (the probe effect concept is described below). All of these examples demonstrate the correlation between the five major issues of deterministic replay: you can not have something without giving up something else [14].

Replay Slowdown

Replay slowdown is often overlooked compared to log size and record slowdown [14]. Deterministic replay tools relying on hardware tend to emphasize smaller logs over faster replay, and deterministic replay tools in software tend to sacrifice replay speed in favor of recording speed ([14], [36]).

Some applications of deterministic replay require the replay execution to be performant (preferably as fast as the recording performance), e.g., intrusion analysis and fault tolerance [14]. We will get back to replay slowdown when discussing those later in Section 3.5. Other deterministic replay tools notable for placing emphasis on replay slowdown can be found in [64], [65], [104], [6] and [36].

It is possible in some circumstances for the replay execution to be faster than the record execution, because system calls that block while waiting for user input may return instantaneously during the replay run since the results can be pulled from the logs.

Implementation Cost

Deterministic replay tools implemented in software have an advantage over tools implemented with hardware support when it comes to implementation

cost [14]. The costs of using dedicated hardware components may not always be acceptable, and it is risky to invest time and money in designing and implementing those if the result may turn out to be unsatisfying. Modifications may need to be done in hardware to low-level caches and their protocols to attain fast recording and accurate prediction analysis, but these areas are particularly time-critical and error-prone components of the processor and thus come with a great deal of risk in changing. Additionally the power consumption goes up [14].

Implementation cost is however considered a critical factor in the applicability of deterministic replay tools and plays a central role, but it is not as important for software-only schemes.

Probe Effect

The probe effect of a deterministic replay tool refers to whether the program execution will be affected by being recorded so that it does not exhibit its original behavior. This is problematic because it may cause erroneous behavior to disappear and therefore conceal it [14]. According to the survey [14], an ideal deterministic replay tool should avoid having a probe effect that may conceal erroneous behaviors of a program execution.

It is mostly deterministic replay tools implemented in software that suffer from the probe effect problem because they tend to impose restrictions on the runtime environment in an attempt to improve recording performance [14]. Deterministic replay tools in hardware on the other hand do not suffer to the same degree because they typically only observe the program execution and do not intervene. The exception is hardware that enforces deterministic parallelism by forbidding potential instruction interleavings to occur in multi-threaded programs. Those kind of tools come with a considerable probe effect [14].

The probe effect can possibly be reduced for software deterministic replay tools by spending more on implementation cost, especially by introducing hardware support [14].

3.4.2 Single-Threaded vs Multi-Threaded Schemes

As the name implies, a single-threaded program uses only one thread to run. Multi-threaded programs on the other hand require multiple threads to run. On a single processor, a multi-threaded program will have its threads

scheduled on the processor one at a time by the operating system. The other threads that remain inactive have to wait for their turn, as only a single thread can run at a time. One of the challenges of recording multi-threaded programs is that threads are scheduled non-deterministically by the operating system, and the order in which they are scheduled must be recorded for us to later be able to replay the same thread interaction with the address space.

The majority of deterministic replay tools work with multi-threaded programs, but IGOR [27], Jockey [98] and gdb's reverse execution feature [2] are some examples of recorders that do not.

Another example is DoubleTake [57]. It is a good example of what can be considered a special deterministic replay tool because it does not record a full program run, only enough to fix crashes. Deterministic replay tools of this caliber tend to be concerned with what happened before the crash occurred, and may as such for instance only record the last ten million instructions prior to the crash. In DoubleTake's case, it allows the program to run at native speed in between checkpoints (the paper refers to this time period as an *epoch*). At each checkpoint, the system verifies that a program error did not occur, and if it did the execution is rolled back to the previous checkpoint and full code instrumentation is activated to allow for detailed recording forward until the error emerges for the second time. Up until the point that the error was detected no recording was being performed and as a result there were significant speed gains (in fact, less than 3% overhead on 14 out of 19 of their benchmarking tests).

3.4.3 Single-Processor vs Multi-Processor Schemes

Deterministic replay schemes can be split into two categories: single-processor and multi-processor. Single-processor schemes (e.g., [27], [95], [25], [100], [111]) deal with recording non-deterministic events such as user input or I/O on a single processor (a more comprehensive list of non-deterministic events were presented in Section 3.3). Multi-processor schemes do the same, but with the additional challenge of predicting or recording the order of inter-processor communication, especially in shared-memory systems ([14], [32]). It has proved to be notoriously difficult to record this non-determinism efficiently, or change it to become deterministic [22], and this has attracted broad interest not only from academia but also the industry (e.g., IBM, Intel, Microsoft, and VMware [14]).

Research in multi-processor deterministic replay is currently the state of

the art as single-processor deterministic replay is considered to be a solved problem ([22], [55]). Many schemes have been proposed but they all fall short in some way. Some approaches require dedicated hardware (e.g., [64], [80], [6]). Others guarantee determinism only if the program is free of data races, only allow the execution to be replayed while recording is in progress, or require an exhaustive search to reconstruct the correct replay, thereby sacrificing the guarantee to be able to replay at all [104] (at least in due time, deducing the orderings of shared-memory operations has been shown to have exponential time-complexity in theory [14]). Often the proposed schemes do not offer time-traveling capabilities or a generic scriptable API, because they are geared towards fixing bugs related to data races or crashes. Interested readers are advised to consult the survey [14] as it contains a wealth of information regarding multi-processor deterministic replay schemes and the current state of the art.

In this thesis we do not consider strategies for multi-processor deterministic replay and as such this area will largely be ignored in the rest of the paper. Information pertaining to multi-processor schemes will only be included to give the reader a better overview of the subject matter when it is deemed appropriate. The focus of this thesis is geared towards single-processor deterministic replay despite it being considered a mature technology ([22], [55], [14]), because the particular area we are interested in appears to have been largely neglected by previous research (user mode without operating system modifications).

3.4.4 Recording Granularity

Most deterministic replay schemes were made to record with instruction granularity. That is to say, the smallest steps recorded are those of processor instructions. Typically a standard tracing tool would record all instructions to the log, but this causes significantly bloated log sizes. Deterministic replay schemes tend to be smarter and use determinism to recover the executed instructions. Only the non-deterministic events have to be logged for this to work; everything else can be deduced from the deterministic property of the program.

Other possible levels of granularity are basic block granularity and function granularity, but more could potentially exist depending on the scope of the particular deterministic replay scheme. An example of a function granularity tracer is *Function Hacker* [62]. It records all inter- and intra-modular

function calls. Clearly this is not as attractive as having instruction level granularity, because a great deal of information is lost, but for quick and easy analysis a tool like this goes a long way considering how accessible it is (it runs in user mode with no operating system modifications). To many developers the luxury of a proper deterministic replay tool is not needed, because it is perfectly possible to do tasks such as call stack analysis, code execution visualization and filtering functions from logs of plain data. In Section 3.5 we showcase the uses of deterministic replay, and it should become clear why it is vastly superior to simple logging. Other examples are StraceNT [42] and NtTrace [38]. Again, both implemented in user mode (not to mention on Windows, a somewhat rare sight), but these record at the library level and system call level instead of function granularity.

3.4.5 Abstraction Levels

The survey [14] goes on to divide deterministic replay schemes into different *abstraction levels*. Abstraction levels are a way to classify interfaces to the outside world from inside the program or system being recorded. When we talk about a deterministic replay tool's abstraction level we consider all levels below it as being the outside world (except for the circuit level which is the lowest level). The abstraction levels presented are as follows (from high to low): library level, system call level, virtual machine level, architecture level and circuit level.

Library Level

A library is a collection of functionality available to the developer to perform well-established, mundane tasks such as copying one string to another or gathering user input from the keyboard. At the library level, it is the user space libraries that are considered to be the interface that must be recorded by the deterministic replay tool. Anything below the library interface is considered to be the outside world, and the recorder is not interested in the internal details of the library, only the returned values and side effects observed. The advantage of recording at the library level is that non-determinism can be isolated to a few shared base functions, but this can not be done for all non-determinism (e.g., shared memory).

System Call Level

The system call level is basically the system call interface: the interface between user space and the operating system. Non-determinism is recorded here by intercepting and logging system calls. System call level logging can record the internals of libraries, because it happens at a lower level. Therefore, it is possible to record bugs within the library that would not be possible to record from outside the library using a library level recording policy. System call level recording is harder to implement because one has to modify the operating system to capture all non-determinism (e.g., thread scheduling). Also, the size of the determinism logs will be bigger because more events are logged compared to logging at the library level.

Virtual Machine Level

The next abstraction level is the virtual machine level. This level facilitates logging below the operating system by making use of the virtual machine monitor (assuming the machine supports virtualization). The virtual machine monitor is a software layer that runs on either the host operating system or the real hardware. A guest operating system is run inside the virtual machine monitor's emulated hardware instead of the real hardware, and as a natural consequence all the events (e.g., system calls, signals, interrupts) made by the guest operating system can be intercepted and recorded (at least in theory). Since thread scheduling happens as a result of these events, by recording them we indirectly solve the problem of recording non-deterministic thread scheduling as well.

Deterministic replay tools at the virtual machine level have the problem that they must record the guest operating system as a whole (e.g., as seen in [25], [53]). As a result, the recorder outputs larger log sizes, and it can be difficult to record only specific processes (which is typically what we want instead of recording the whole system [36]). This particular problem has also been identified in other research areas (e.g., malware analysis) as the *(virtual machine) semantic gap* ([7], [54]). Shortly put, even though the virtual machine monitor (or in general, any full system emulator) has unrestricted access to all the low-level data of the guest operating system, it is not immediately clear what raw data corresponds to what objects within the guest system.

Virtualization needs to be supported to be able to employ a virtual ma-

chine monitor, which may disqualify some older systems or systems restricted by hardware limitations (e.g., embedded systems). Virtualization is, however, supported on most newer systems, but it can still be time-consuming to set up a virtual machine deterministic replay tool and as such be harder to deploy. Anderegg [3] considers these obstacles to be factors reducing the deterministic replay scheme's practicality.

A well-known virtual machine deterministic replay tool that experienced public usage was VMware's Record and Replay feature built into VMware Workstation 6.0 [35] (referred to in the literature as ReTrace [111]). Because VMware Workstation is a mature product, its set up process can be estimated to be more user friendly and easier than the majority of other virtual machine deterministic replay tools, thus alleviating the aforementioned problem by allowing users easy access from the built-in menu. Despite its potential for widespread common use, this feature was discontinued in VMware Workstation 8 [93] (the lack of interest was in fact the reason for its downfall).

Architecture Level

The architecture level is the first abstraction level that is not implemented using software. This is the level where hardware-assisted recording comes into play. Since most non-deterministic issues have already been dealt with at the higher levels it is for the most part shared-memory operations that is the focus of this level. Deterministic replay tools at the architecture level are normally not widely deployed because of their high implementation cost. They do however excel at having low recording overhead ([14], [36], [3]).

As a side note, Intel has in recent years begun to include an efficient tracing feature in their newest processors that they have called Intel Processor Tracing™ [87]. It offers low-overhead tracing in commodity processors and will most likely become a cost-effective way (compared to the typical dedicated hardware seen in this area) to achieve lightweight replay capabilities. We say lightweight because it is not a deterministic replay scheme and only logs plain data. Even though this feature is still in its infancy, it has been integrated as an option in gdb's record and replay feature ([87], [20]).

Circuit Level

The circuit level is out of scope for this paper but was included for the sake of completeness. Summarized, it deals with very low-level non-deterministic

asynchronous events (on the order of 10^{11} per second!), and as such the research focus is on making these events deterministic instead of trying to log them. This is similar to what we have seen before happening at the architecture level; if it is difficult to record something due to volume, try to force it to become deterministic instead, so recording can be avoided altogether.

Summary

The abstraction level of a deterministic replay scheme has a significant impact on its design choices. Schemes at a higher abstraction level can ignore details of the lower levels, but will usually have more overhead since they lack the power to do things in a more efficient way. For instance, a user mode deterministic replay tool residing at the system call abstraction level will have a hard time recording the process's thread scheduling, because from the perspective of the recorder these are non-deterministic events. If the deterministic replay tool instead modified the operating system to notify it on thread switches, less performant designs would not be necessary.

Due to the diversity in deterministic replay applications there is no universal solution that performs well in all scenarios.

3.4.6 Fidelity

A less frequently mentioned characteristic of what makes a good deterministic replay tool is the *fidelity*, or *faithfulness* of the system. In fact, this characteristic may very well be the most important of them all. The reason it was not mentioned earlier is because it is naturally implicit in the definition of deterministic replay. According to the survey [14], fidelity can be described as the degree to which a deterministic replay scheme recovers the details of the execution of a program. In other words, what happened during the original recorded program execution must also happen when the recording is replayed. What this means is that all changes that were made to the execution context, such as changes to register values or memory contents, must be faithfully reproduced during the replay execution. Clearly, if this was not the case, how could we trust that what we see in the replay actually happened in the original program execution?

The survey [14] then goes on to further categorize their definition of fidelity. The most frequently seen type of fidelity is called the *logical determinism*. This type of fidelity guarantees that each variable access seen in the

record run corresponds to a unique variable access in the replay run. Both accesses must produce the same result.

However, the fidelity granted by the logical determinism is not enough because it does not cover some time-related events. It does not say anything about the time interval between one event and the next. For this reason another type of fidelity needs to be introduced: the *timing determinism*. Simply put, the timing determinism holds the same guarantee as the logical determinism, but with the additional requirement that each variable access in the record run must take place at the same time point in the replay run relative to the beginning of the execution.

3.4.7 Apples and Oranges

Deterministic replay tools span multiple different categories, and it can be tricky to classify each of them properly. It is not uncommon that multiple tools share similar traits, yet at the same time are vastly different in other aspects. This makes it challenging to compare tools, even if they fit the same set of criteria.

For instance, consider that we were to compare two user mode deterministic replay tools, rr [82] and iDNA [8], on the basis of performance overhead. In this scenario, rr would be the victor as its paper presents lower performance overhead than iDNA's paper. However, rr does not support multi-processor recording, which means that its superior performance may be inflated and would not be up to par if it was able to record parallel programs on multiple cores like its competitor. iDNA also has better log compression rates than rr [3]. Does this mean that iDNA is a better deterministic replay tool overall compared to rr, despite rr having lower overhead?

All deterministic replay tools, no matter the category they are classified to belong in, tend to compare against each other this way. Even for seemingly similar tools, like in the example above, there may be significant differences that are not apparent at first glance. It is therefore important to keep in mind that every tool has its strengths, but that no tool is able to fulfill all criteria and that there is no panacea. If there was, this thesis would not exist. The arms race of which deterministic replay tool is the best is a game of *trade-offs*, where the tool that best fits the problem domain wins. With that said, tools that are easily deployed in industry are considered to be more successful than those that are not on an overall basis according to the survey [14].

3.5 Applications of Deterministic Replay

The use cases for deterministic replay span a wide range of domains, e.g., hardware testing, computer architecture, operating systems, parallel computing, programming languages and program verification [14]. Example applications include postsilicon debugging [14], processor architecture simulation [80], performance prediction [14], online program analysis [14], intrusion analysis [25], fault tolerance [95], debugging [53] and automated offline program analysis [24]. We are not particularly interested in the hardware related applications in this thesis and will focus on the software applications of deterministic replay.

3.5.1 Online Program Analysis

Online program analysis is a technique in which a running program is being monitored to detect errors as they happen. By checking the program's data flow and control flow, it is possible to determine whether something unexpected happened and terminate the program [14]. A typical example is *taint analysis* [14], a technique where data flowing into the program from outside sources (such as user input) is marked as being *tainted*. Tainted data is considered a security risk as it may influence internal program state in ways that were not intended, and the data is therefore monitored as it spreads throughout the program. Any new data resulting from an operation with tainted data is itself considered tainted, and if the tainted data is used in sensitive ways (e.g., as part of a SQL query) the taint analyser will raise an error.

Online program analysis is widely regarded as an effective technique to detect bugs and defend against security attacks on-the-fly, however, the techniques are often heavyweight with consideration to runtime overhead ([14], [31]). This is especially true for analysis performed with instruction granularity, and it is not uncommon to experience more than 10x slowdown on commodity computer systems [14].

By using deterministic replay, we can perform online program analysis on spare cores, thus offloading the heavy work from the main program execution. This requires that the recording slowdown is not too significant because it must be supported during the whole lifetime of the program run [14]. Additionally, replay slowdown can not be too remarkable and should not be bigger than the record slowdown, because then the replay run would

never finish in time to catch up with the original program run. To exemplify this, if the replay slowdown was ten times that of the record slowdown, we would need ten times more cores to finish replaying in time [14].

Log sizes, on the other hand, are not important, since only recent log slices of the execution are necessary to perform the online program analysis [14]. It would therefore be possible to use a cyclic buffer as the log output.

3.5.2 Fault Tolerance

Computer systems that require a high degree of uptime may use deterministic replay to fulfill that goal [95]. By recording the system's execution until the system crashes, it is possible to recreate its original state before the crash and allow it to keep running, thereby maintaining the necessary level of uptime. This is a case where replay speed is more important than usual, because if the reinstated system is to get up to speed in an acceptable amount of time, the re-application of recorded events needs to be quick. Taking snapshots of the system state at regular intervals may help speed up this process, because otherwise every event must be re-applied by deterministically executing from the beginning.

Deterministic replay can also be used to *detect* when a system fault occurred. For example, ExtraVirt [58] runs two identical systems on two virtual machines, one acting as a replay of the other, and as such all non-deterministic events seen in the first machine are reproduced in the second machine from the first machine's logs. Snapshots are taken periodically of the system state on both machines and then compared. If the two snapshots are identical a new snapshot is taken and the old snapshot is discarded. In the event that they do not match it can be assumed that a fault occurred and both systems need to be rolled back to the previous snapshot [14].

3.5.3 Intrusion Analysis

When a computer system has been compromised by an attacker, the system can be considered tainted. Operating system files and personal files may potentially have been tampered with, and it can be challenging to determine what steps the attacker took once access had been gained into the system. Normally the system administrator would perform post-attack analysis and consult the operating system logs for clues. The defect in this strategy is that

the operating system may no longer be trustworthy if its kernel was compromised. The attacker may have deleted or modified the system logs or disabled further logging. The system's *integrity* would have been compromised [25].

Another problem is *completeness*. Because the operating system has to be constantly logging, it is not feasible to log too many details for both space and performance reasons, and often only certain system events get logged. Once the system has been compromised the system administrator is left to guess what occurred during the break-in based on the limited amount of information found in the log [25].

By moving the operating system into a virtual machine and logging below the level of the virtual machine it becomes possible to overcome these obstacles. Because it is difficult to attack the host operating system from the guest operating system, the determinism logs can be considered loyal, and log integrity is thus preserved. Logs may also contain more information (logically, not physically), because the logging policy of a deterministic replay tool is vastly different to that of an operating system. For instance, useful details about the breach may not be logged at all, but by logging the non-deterministic events the whole system can be replayed, and it is then possible to reproduce the information the system administrator is interested in without actually having to log it inside the operating system itself. A deterministic replay tool like this was presented by Dunlap et al. in [25].

Despite shielding the intrusion analysis component by hiding it outside of the virtual machine, it is still important not to impose too much recording overhead, because it will have an effect on the throughput of the system. Recording has to be active all the time to be able to replay from a point before the intrusion happened. Additionally, it is possible for malware to detect when it is being analysed under a microscope if the execution speed is slower than the expected native speed [88]. The malware may then alter its original execution path to evade analysis without the analyst knowing, and this could leave the system untouched. This would be a good thing for that particular system's health, but bad for all the other affected systems where the malware took its original execution path, because the analyst was unable to determine the inner workings of the malware and thus unable to publish a patch.

Some systems need to stay online for months or even years, so it is crucial that log size stays small. On the other hand, replay slowdown is not important for intrusion analysis since having to replay is quite rare (only when the system was compromised) [14].

3.6 Debugging Using Deterministic Replay

Probably the most prominent use case for deterministic replay is debugging ([53], [14]). There are multiple ways to increase debugging efficiency and effectiveness with the support of a deterministic replay scheme. Engblom makes a distinction in his paper [26] between the different types of debugging that are possible with the help of deterministic replay, and we will discuss each of them below.

3.6.1 Cyclic Debugging

With a deterministic replay tool in place, it becomes possible to employ traditional cyclic debugging without all the caveats of non-determinism. The normal workflow of cyclic debugging is to re-run the program being debugged whenever the need arises, typically to use newly gained information at an earlier stage in the program execution where it would have more value. Debugging cyclically is a risky affair: whenever the program is restarted there is a chance the program's execution path may differ from the original run due to non-deterministic factors. This is particularly true for parallel programs and programs with an otherwise high degree of non-determinism.

By using a deterministic replay tool to debug the erroneous program this risk is eliminated. A bug that would normally manifest itself once every thousand program runs can be reproduced faithfully on *every* replay run and diagnosed properly, once and for all. The ability to replay this way greatly alleviates the pain associated with debugging non-detereministic programs, and it is especially apparent when dealing with race conditions or heisenbugs.

A *heisenbug* is an umbrella term used to describe a class of bugs that seem to magically disappear once you go looking for them. Another name for such a bug could be a *non-deterministic bug*. Heisenbugs are hard to reproduce because they do not necessarily just depend on input to the program, but also on the environment in which the program is running (e.g., operating system, debugging environment, hardware configuration) [80]. A race condition is a type of heisenbug that occurs when resources are accessed in an order not expected by the developer (e.g., non-deterministic thread interleaving). It is arguably one of the most difficult classes of bugs to fix in modern computing due to the increase in multi-threaded programs running on multi-processor systems, and as a result a great deal of research has therefore been devoted to detecting, avoiding and eliminating race conditions, ranging from static

analysis to on-the-fly dynamic analysis. Both of these problems can be solved by using a debugger that has the support of a deterministic replay tool.

In essence, cyclic debugging is built on the assumption that multiple executions of a buggy program should exhibit the same erroneous behavior. As demonstrated, this assumption does not hold in practice [14].

3.6.2 Time-Travel Debugging

Cyclic debugging coupled with deterministic replay is a clear improvement to the traditional debugging experience, but it still only allows the developer to debug in the forward direction. The developer has to restart the replay to be able to go back to an earlier point in the program run, but can do so without fear of non-determinism changing the execution path. A *time-traveling debugger* on the other hand, as the name implies, makes it possible to move forwards *and* backwards ([53], [56], [8], [80], [26]). In a time-traveling debugger all standard debugging functionality is available to the developer, but with the additional feature that most common operations can be performed in reverse. For instance, it is possible to single-step or run the program backwards, thus allowing the developer to place breakpoints and watchpoints at arbitrary points in the execution history of the program. Due to the apparent productivity gains in using a time-traveling debugger it has long been considered the holy grail of debugging.

"Time" in this context does not necessarily refer to wall-clock time as most time-traveling debuggers use the concept of *execution time*, or *number of instructions executed*. This makes it easier to precisely pinpoint the exact location of the execution at an arbitrary point in time, something which would have been difficult using wall-clock time due to the sheer number of instructions executed per second and the fact that different hardware may offer different such numbers. Execution time is a simpler format that is more in line with how a machine would see the execution (as discrete steps).

Once a program's execution has been recorded, the rush of needing to perform certain tasks at certain time intervals while debugging disappears, because all non-deterministic events are reproduced from the log and will never change no matter how much time the developer takes to perform the analysis. This can greatly reduce the stress experienced by the developer while debugging certain tasks that rely on precise time-interaction.

An example of the usefulness of a time-traveling debugger is making an educated decision based on information only known in the future, such as

the result of an operation. A normally impossible task suddenly becomes notoriously easy as we can run forwards until we hit our breakpoint, collect the result of the operation, then put a breakpoint on the destination that is supposed to be in the past and simply run backwards until the breakpoint is hit. We can then make the decision.

Arnold, an "eidetic system" (authors' words) created by Devecsery et al. [22], takes this idea to its full potential. It allows the user to look back into the past in the form of queries. One of their examples goes like this: Imagine you realize that you cited the wrong paper. You write a query, asking whether any process or thread in the past was involved with citing said paper. You find and reconstruct the browser page where you downloaded the paper and notice it happened to be from the wrong conference. You then write a new query, asking into the *future* about where the citation was used so that you can undo it. This same example can be extended to how malware spreads throughout a system, and how all changes made by the malware can be detected and undone.

One of the earliest and most well known examples of a deterministic replay tool that allows time-travel debugging is the system created by King et al. in [53]. It is implemented as a virtual machine that is able to record a full operating system run, thereby eliminating the traditional issues associated with operating system debugging. Operating systems are hard to debug cyclically due to their inherent non-determinism (all the typical problems associated with non-determinism, but adding hardware communication etc.), and the fact that the operating system often has access to the debugger's memory and may therefore corrupt it [53]. The tool's replay interface has been integrated into the gdb debugger, allowing the developer to use most of gdb's standard functionality, in addition to the specially added reverse operations. This is an approach also seen in other time-travel debugging implementations, for instance rr [82] and gdb's own record and replay feature [21].

Checkpoints

Typically, to achieve time-travel debugging, checkpoints need to be taken at periodic intervals during the recording phase. It is not strictly necessary, but it will significantly speed up the debugging process if we can avoid having to replay the whole execution from the beginning. A checkpoint mainly constitutes the visible system registers and the memory state. In order to

improve performance, full system checkpoints should be avoided, and only values that changed since the last checkpoint should be saved instead (this is referred to as *copy-on-write* in this context) [80].

Many computer operations are irreversible and destroy information, e.g., XORing a register with itself or writing a new value to a memory location. It is not possible to take the state after such an operation and infer the state before the operation [26]. Considering that it is fundamentally impossible to reverse the execution of a computer program, we are limited to executing forwards. The operations that seemingly work in reverse are actually executed deterministically from the most recent checkpoint prior to the target of the operation. For example, to execute backwards we first place a breakpoint at an instruction we know was executed in the past. We then execute deterministically in the forward direction from every checkpoint, starting at the most recent and working our way backwards. Once the breakpoint has been hit we stop the search and return control to the user [80]. From the developer's perspective, it will appear as if the execution truly happened in reverse. For the operation to appear instantaneous to the developer, intermediate checkpoints need to be created at short intervals. This results in bigger logs, and it is therefore a trade-off between debugger responsiveness and recording efficiency [80].

The alternative to using checkpoints is to log everything. For some embedded systems this is feasible due to tracing capabilities in hardware, but for systems implemented in software the time and space overheads tend to be too high to be practical. It is therefore preferred to save as little information as possible with a checkpoint mechanism (only the information that can not be reconstructed), and then execute deterministically from the saved state in order to reproduce past state [26].

Two early examples of deterministic replay tools making use of checkpoint mechanisms can be found in [27] and [9].

3.6.3 Automated Analysis

Taking it to the next level past time-travel debugging, instead of simply debugging manually using the replay mechanism, some deterministic replay tools (e.g., [8], [24]) offer time-traveling capabilities in the form of scripting by offering an API. Such features allow for deeper analysis because a human does not need to be present, the analysis can run much faster, and scripts can be made generic and reusable which allows them to be used on multiple

unrelated program recordings. Depending on the deterministic replay tool's log compression abilities, it may be possible to record a program execution once and upload it to the Internet so it can be shared with others. Additionally, different scripts can be run on the same recording on different machines, allowing for very deep analysis from multiple angles at the same time.

A Real Example

An embodiment of this idea of automated analysis is the deterministic replay tool created by Dolan-Gavitt et al. called PANDA (Platform for Architecture-Neutral Dynamic Analysis [24]). As can be guessed from its full name, PANDA is architecture neutral. It is implemented as an extension of QEMU, a popular software emulator which supports emulation of multiple architectures, and can as a result work out of the box with thirteen different processor architectures. PANDA was designed with reusability and sharing in mind, and as such it offers a plug-in facility that lets its users write scripts (or "plug-ins" as they call them) that can be applied to the replayed executions. Multiple such plug-ins already exist and can easily be used with new recordings unknown to the plug-in author. The plug-in architecture allows plug-ins to share functionality, opening up the possibility of combining multiple plug-ins into one. This is useful because it lets us avoid having to explicitly hard-code multiple behaviors into a single plug-in as one would normally have to, and instead allows us to rely on the software principle of composition. Because PANDA is architecture-neutral, a plug-in will work with no modifications on every processor architecture supported by QEMU, this thanks to the LLVM IR used internally in QEMU which decouples it from a specific processor architecture. As mentioned, PANDA allows its recordings to be uploaded to the Internet and shared with others thanks to its effective log compression. The downside of relying on QEMU is that it can be slow [110].

3.6.4 Characteristics of a Deterministic Replay Debugger

To offer a robust and flexible debugging experience, certain criteria need to be fulfilled by the deterministic replay debugger. Most important is the probe effect. If the probe effect is too strong the erroneous behavior that is to be debugged may not re-occur while replaying [14].

Some bugs may only emerge after executing for several days, therefore record slowdown is also an important factor. Not only will a significant performance overhead have an effect on the overall throughput of the system, but it may also slow down the replay process. Since the program execution must be deterministically replayed step-by-step from the last checkpoint (if any) it may take a considerable amount of time to reach the point where the bug occurred [14].

Log size and replay slowdown are not as important factors in this case, because most deterministic replay tools employ a checkpoint recovery mechanism. By periodically saving checkpoints we can discard the logs captured before the recent checkpoint, and by using checkpoints we can immediately skip into the replay close to our desired destination [14]. Regardless, both of these would be nice to have in a deterministic replay tool as they improve the overall user experience and save space and time.

However, one scenario where small log sizes would be preferred is that of remote debugging. As long as the recording overhead is acceptable recording can always be turned on, and when a bug manifests itself or the program crashes the user can submit the recorded logs to the developers. The developers can then analyse the logs using deterministic replay (even multiple developers at the same time). Since the program execution is in a serialized format it should be easy to send it across to the developers, but this does require the log size to be somewhat generous [56] (for instance, Microsoft's time-traveling debugging scheme will not allow the user to upload logs surpassing 2 GB [52]). Similarly, beta testers can e-mail a debugging session to developers even if they do not have the source code [56]. Recording slowdown is not as important if the bug happens to be deterministic in nature. All the user then has to do is restart the program and induce the bug. Lastly, it is easier to send in a determinism log than writing a bug report [80].

Chapter 4

Deterministic Replay in User Mode

We have so far looked at the different categories of deterministic replay tools and their application areas. In this chapter, we will narrow our focus to that of deterministic replay tools that live in *user mode*. Both recording and replaying in user mode can be considered challenging due to the limited amount of options we have at our disposal to observe and manipulate the program's address space as it executes, something which deterministic replay tools residing at other abstraction levels may find to be trivially easy in comparison because they do not lack these options.

In this chapter, we will first discuss what it *means* to be recording in user mode. As it stands, the terminology can be considered vague, therefore we aim to make the conflicting definitions distinguishable to allow us to narrow our focus. Next, we will introduce the concept of user mode virtual machines, which is an extension of the previous topic. We will then introduce the concept of instrumentation and how it is a central part of many deterministic replay tools. Lastly, we will look at the challenges deterministic replay tools face in user mode and demonstrate how they become amplified in Windows.

4.1 Operating System Modifications

A program that is being recorded by a deterministic replay tool in user mode records all non-deterministic events (and possibly deterministic, too) within the scope of the program's process address space. For multi-threaded programs this includes recording all threads and how they execute in relation to each other throughout the lifetime of the program execution.

In user mode, we are confined within the process's address space. The lowest level we are able to record at is the system call level where the system call interface creates a bridge that leads into kernel mode. The effects of interrupts and other asynchronous events are restricted to what is visible in the address space; we simply are not interested in what goes on inside the operating system behind the scenes.

If we do not care about operating system internals, then how come so many user mode deterministic replay tools happen to rely on exactly this information? How can a deterministic replay tool claim to be implemented in user mode, yet depend on having the operating system on which it runs to be modified? We do not know the answer to that question, but what we *do* know is that we need to clearly define what constitutes deterministic replay in user mode to avoid any further ambiguity.

4.1.1 Upsides

The vast majority of user mode deterministic replay tools modify (or extend) the operating system to get the job done (to name a few: IGOR [27], Respec [55], Cyrus [36], Echo [43], Flashback [100], iDNA [8], and the tool created by Russinovich et al. [95]). In fact, we cannot remember having seen a pure user mode deterministic replay tool at all that satisfies modern criteria of being able to record multi-threaded programs with low overhead.

There is nothing inherently wrong with this, because it is still all implemented in software and most likely has a considerably lower implementation cost than many other deterministic replay tools. By modifying the operating system, we also get access to improvements such as less recording slowdown, because we can simply hook into the operating system routines that handle what we need and publish notifications to user mode instead of being confined to user mode and thus having to implement solutions with a higher complexity (and consequently, higher overhead). Additionally, some operating systems facilitate easy deployment of operating system modules (drivers)

or extensions, thus allowing the required modifications to be shipped together with the user mode executable (the deterministic replay tool).

4.1.2 Downsides

The problem appears when the operating system resists this kind of malleability. Not every operating system permits its users to extend or otherwise modify its core. For instance, Windows 7 64-bit which is the platform on which our proposed user mode deterministic replay tool is designed to be implemented, does *not* permit this in an easy manner. It is possible to write kernel drivers (a driver is the same as an operating system module in Windows terminology) in Windows, but it is not possible to install them without having the driver cryptographically signed by Microsoft ([68], [69]). An exception to this is signing the driver yourself with a test certificate, but to install the driver you will still need to configure the operating system at boot time to enable loading drivers signed for testing [75]. This is the first major downside to relying on modifying the operating system: **deployability**.

The other downside is the act of modifying the operating system itself in the first place. Kernel drivers are notorious for being dangerous, that is, when code is running in kernel mode there are no restrictions on what it can do, and as such it is possible for a bug to corrupt the operating system's data and cause other instabilities. On Windows, if detected, this would lead to the infamous *Blue Screen of Death* in an attempt to avoid damaging the computer [67]. This is the second major downside of modifying the operating system. It breaks the user mode **safety guarantee**. As an attempt to keep Windows more reliable and safe from malware Microsoft has also added protection against kernel patching. This means that even if we were using a driver we would be very limited in what we would be able to do [67].

Another downside worth mentioning is the level of complexity inherent in developing a kernel driver. Writing a kernel driver is completely different from your standard run-of-the-mill user mode program as there are many difficult problems that need to be taken into consideration to successfully implement one (e.g., properly securing your code against being interrupted at any moment, safely interacting with memory, and so on). It is therefore a high-risk endeavor (as is evident from the previous paragraph) that should not be taken lightly without the necessary programming experience.

A special exception to all of these issues is the user mode deterministic replay tool iDNA [8]. iDNA was developed at Microsoft and could therefore

enjoy the luxury of both having its driver signed (because Microsoft is the certificate authority) and its code properly written by kernel experts, thus avoiding all the downsides of using a driver. The reason iDNA is mentioned here is because it is one of the aforementioned user mode deterministic replay tools that rely on operating system modifications. iDNA's runtime engine Nirvana can be configured to either rely on direct operating system modifications or a kernel driver [8]; in other words, it is not a pure user mode implementation.

4.1.3 Conclusion

Where do we draw the line between what constitutes a "proper" definition of user mode in relation to deterministic replay and what does not? For example, as was described in the previous section, iDNA [8] relies on operating system support either directly or through using a driver. Even though by our definition this is a clear violation of the user mode contract (as is evident from the detrimental effects of relying on a driver as previously described), other researchers may still consider this to be a user mode implementation (e.g., [3]), so we are left with no clear consensus. This distinction between definitions may potentially vary from operating system to operating system depending on the limitations of said system.

Also, where do we draw the line on how strict such a definition should be? For example, rr [82] is a deterministic replay tool that claims to be implemented purely in user mode, however it indirectly does not fulfill this criterion because it uses hardware performance counters, a feature that is reserved for kernel mode (the model-specific registers can only be edited with the *rdmsr* and *wrmsr* instructions, both which are only usable in kernel mode [18]. To record the non-deterministic *rdtsc* instruction, rr also relies on a feature that disables that specific instruction and instead traps to the operating system on its execution (this will be elaborated on in Section 4.4.2). The reason rr can still be considered a user mode deterministic replay tool is because the access to these special features happens through a user mode API exposed by the operating system (Linux), so technically it is still a user mode implementation. On the other hand, nothing of the sort can be found on Windows, effectively disqualifying the use of hardware performance counters and trapping to the operating system for the *rdtsc* instruction. For this reason, we cannot agree with the (additional) claim made by rr that its implementation runs on stock operating systems (or at least the implication

that its design can be transferred to other stock operating systems) because Windows can clearly be labeled as a stock operating system, yet fundamental design choices made in rr do not work on Windows.

This example goes to show that designs published from research on user mode deterministic replay tools may not be applicable to other systems that rely on the more strict definition of user mode (no operating system modifications), and it would therefore be preferable to distinguish between the two definitions to make it clear which is being used for that particular research.

To conclude, we can say that deterministic replay tools that require modifications of the operating system are not easily usable in practice until support for deterministic replay techniques is integrated into the operating system by default [3].

For the rest of this thesis, we will consider deterministic replay tools that claim to be implemented in user mode to refer to the more relaxed definition of recording and replaying in user mode. That is, it may imply that operating system modifications are necessary to successfully use the tool. With regards to our own proposed tool, we will be using the definition of *pure user mode* (and it will be referred to as such) which prohibits any form of operating system modification.

4.2 User Mode Virtual Machines

Following in the footsteps of the previous section, we will next discuss deterministic replay tools based on virtual machines in user mode. A distinction needs to be made between actual virtual machine software such as VMware [1] and VirtualBox [2] and *managed runtimes* because we will only discuss the latter. We consider the former to fall within the abstraction level of virtual machines (see Section 3.4.5 on abstraction levels) and as such it is out of the scope of this discussion (which is about user mode).

User mode virtual machines or *managed runtimes* are well-known pieces of software in the industry that run exclusively in user mode. Examples include (with examples of deterministic replay tools for that managed runtime in parentheses) the Java Virtual Machine (JVM) for Java-based programs ([56], [101], [1]), the Common Language Runtime (CLR) for .NET-based programs ([5]), and emulation software like QEMU and Unicorn ([24], [86]).

[1] https://www.vmware.com/

[2] https://www.virtualbox.org/

We remind of the distinction between emulation and virtualization as we are here only talking about pure software emulation in user mode (see Section 2.2.3 on emulation). Another class of managed runtimes often seen in this area is that of web-based browser runtimes such as JavaScript engines or Flash engines.

4.2.1 Recording in User Mode Virtual Machines

The way recording is done in a user mode virtual machine is quite different than in native binary programs. The virtual machine typically abstracts away the underlying operating system resources and the operating system API from the program, limiting system interaction down to a significantly smaller surface. This limited surface can then be instrumented to capture all non-deterministic events entering the program's space from the outside. For instance, some managed runtimes have their own thread scheduling mechanism and the program's thread switches are determined by the virtual machine and not the operating system, therefore it can easily be instrumented.

The authors of TARDIS [5], a deterministic replay tool in C#, is an embodiment of a tool that uses such techniques. TARDIS modifies the standard library and the authors point out that it is quite risky to do this, because missing or incorrectly modifying an API may have unwanted effects on the recording. It is therefore preferred to locate the root APIs that all the other APIs eventually delegate to and instead only modify these, thereby limiting the surface area and additionally lowering implementation costs.

Similar to this is instrumenting the bytecode utilized by those virtual machines directly. Bytecode tends to be less fine-grained than native instructions in their instruction sets, therefore it is easier to instrument bytecode than native code due to the reduced number of instructions available, and because the semantics of each instruction may be more self-contained and thus easier to reason about. Since user mode virtual machines are implemented in user mode, it is also fully possible to modify them either by recompiling the virtual machine from public source code, or by reverse engineering the virtual machine and installing instrumentation. Software emulators like QEMU may even offer explicit access to every executed basic block or instruction in the form of callbacks. This kind of flexibility made it possible to perform early experimentation on the ideas surrounding time-travel debugging before binary instrumentation frameworks like Pin [59] were easily accessible. An example of this is the Omniscient Debugger [56] implemented in Java.

Despite having a great deal of flexibility, not every user mode virtual machine is able to record all non-determinism (e.g., I/O and DMA), or parallel programs.

4.2.2 Conclusion

Somehow this sounds all too familiar to what was discussed in the previous section. If we were to consider user mode virtual machines as miniature operating systems things would start to make more sense as we could consider this to be another form of operating system modification. Can we consider deterministic replay tools based on user mode virtual machines to fall in under the definition of pure user mode? We are inclined to say yes, because there are no explicit operating system modifications going on.

Because of the apparent simplicity and flexibility in recording user mode virtual machines (relative to recording native programs), this thesis was from the beginning aimed at recording *native* user mode programs in binary form. We therefore need to once again redefine the context of how we use the term "user mode" throughout the rest of this paper, and continue by narrowing our definition further to that of native binary programs.

4.3 Instrumentation

Many deterministic replay tools utilize instrumentation (ref. Section 2.2.2) to some degree, especially in the recording phase, and instrumentation is therefore a central topic in deterministic replay. Instrumentation offers us a great deal of control over the program we are instrumenting, and it can be helpful in recording information that may otherwise be challenging to acquire without instrumentation. However, this control comes at the cost of increased complexity and runtime overhead.

4.3.1 Instrumentation in User Mode Virtual Machines

As was previously discussed, instrumentation is a commonly seen technique in user mode virtual machines since instrumentation points often already exist as part of the architecture of the virtual machine. This allows programs to utilize instrumentation wherever they may need it through an accessible

API. Not only that, but bytecode is more easily instrumented than native code due to its often higher level of abstraction.

4.3.2 Instrumentation in Native Programs

For native programs instrumentation happens either at compile-time or at runtime.

Static Instrumentation

Compile-time instrumentation or *static instrumentation* is easier to implement and may possibly perform better than dynamic instrumentation because static analysis can be used to identify potential bottlenecks, and optimizations can be made by the compiler since it has access to type information, something the dynamic instrumentation tool does not. Instrumentation support can be part of the development environment and be added automatically when the program is compiled. This is useful for in-house debugging as instrumentation can later be disabled for production code to reduce performance overhead (which is the preferred option [26]), but instrumentation is still available for debug builds. Another level of instrumentation similar to static instrumentation is modifying the operating system libraries [26].

An example of a deterministic replay tool leveraging static instrumentation is the Live Recorder from Undo [102] which allows the developer to use time-travel debugging with their complementary debugger UndoDB. Their average recording slowdown is 2-3x [103]. Another example is the tool created by Boothe in [9]. Lastly, Russinovich and Cogswell [96] employed a software instrumentation counter [63] to preprocess compiled assembly files, which can also be considered as a form of static instrumentation as the instrumentation does not happen at runtime.

Although useful, static instrumentation is inflexible in that it does not allow us to record already compiled binaries that do not come with instrumentation. For this reason static instrumentation at compile-time can be thought of as a somewhat limited technique when it comes to deterministic replay.

Dynamic Instrumentation

Dynamic instrumentation (more specifically binary translation) solves the problem of instrumenting pre-compiled binaries by instrumenting the program as it executes. This comes with a performance penalty because every basic block must be analysed to decide whether or not to apply instrumentation. When instrumentation is applied, it is necessary to analyse the instruction sequence found within the basic block and rearrange and modify it to comply with the new, instrumented version of the block.

Because deterministic replay tools often want to record memory accesses, the instrumentation process becomes very expensive since memory accesses are frequent in assembly code (x86), and instrumenting each one of them is time-consuming. Despite this slowdown, dynamic instrumentation can offer a very precise level of recording granularity; basically every instruction that is executed can be analysed and thereby logged if it fulfills the required criteria.

For instance, many deterministic replay tools struggle with recording shared memory and DMA. This is because it is possible for external sources (e.g., other processes or hardware devices) to modify the shared memory, and the memory is therefore not deterministic because it may change at any time without having been explicitly modified by the program itself. By instrumenting the code as it executes the deterministic replay tool can observe when memory is accessed by the recorded program and log each access individually. If a memory address is accessed twice and an external source silently updated its value in between the program's own accesses, this change will be detected by the recorder and appropriate action can be taken.

Recording all memory accesses produces massive logs, and it is therefore better to only log accesses that cannot be predicted or reproduced with determinism. For example, iDNA [8] only logs memory reads that cannot be predicted by comparing the current read to the previous read from the same address. No log entry is created if the values are identical, and the *prediction count* for that memory address is incremented. During replay repeated reads of a given memory address will always return the same value until the prediction count of that address has been exceeded (the prediction count is also incremented during replay). When this happens a new value will be assigned to that memory address from the log. A similar approach is also taken by BugNet [80].

Dynamic Instrumentation Cost

Performance The main problem with dynamic instrumentation is that it is slow. This does not facilitate efficient deterministic replay in practice [3]. The authors of iDNA [8] report a slowdown of 12x-17x when recording CPU-intensive programs with their framework. For less intensive programs the overhead is smaller (as low as 5x), and it is feasible to leave the recorder running in the background. PANDA [24], another deterministic replay tool based on instrumentation, reports 2x recording overhead to that of normal QEMU execution using software emulation (which, as described in Section 2.2.3, is quite slow compared to native execution).

The difference between iDNA and PANDA is that PANDA takes the emulation approach whereas iDNA's instrumentation is based on dynamic binary translation. Emulation is usually slower than binary translation even though the binary translation approach analyses and instruments basic blocks. This is because when emulating, all the executed instructions not only have to be instrumented (unless only basic blocks are being instrumented), but the instructions themselves must also be emulated in software, whereas the binary translation approach allows the program to run natively once its instrumentation phase has been completed.

Despite dynamic instrumentation being slow in general, it could still be worse. Debuggers with built-in tracing capabilities (e.g., gdb ([23], [53], [2]) and OllyDbg [112]) often employ rather primitive ways to trace the program being debugged, the most common one being to use the processor's trap flag (x86). When the trap flag is active the processor will not execute instructions normally but will instead issue an interrupt (int 1) on every instruction, essentially single-stepping. This interrupt is then picked up by the operating system's interrupt handler for that interrupt vector, which for int 1 is typically the debugger's single stepping routine. The debugger then logs the instruction and all necessary context and continues running (in reality single-stepping). This is an extremely slow way of tracing the program, possibly with a factor of several thousand. gdb attempts to ameliorate this by offering alternatives: Intel Processor Tracing (mentioned in Section 3.4.5) and Branch Trace Store which only records branches and no data ([20], [2]). Not all debuggers are this lucky and are left with the aforementioned method of single-stepping. This should demonstrate that dynamic instrumentation is slow, but still quite fast.

Complexity Implementing a dynamic instrumentation framework is a rather complicated task and most deterministic replay tools therefore use existing frameworks to reduce their implementation cost. One of the most well-known frameworks for dynamic binary instrumentation is Pin [59] which was mentioned in Section 2.2.2. Many deterministic replay tools use Pin internally as their instrumentation engine, some examples being BugNet [80], PinPlay [84], IDA Pin Tracer [97] and Qira [86] (Qira can also use a QEMU backend [37]). Others implement their own instrumentation framework to more closely fit their needs, e.g., iDNA's Nirvana instrumentation engine [8].

4.4 Challenges of Deterministic Replay in User Mode

To achieve deterministic replay, non-deterministic events must be recorded. Deterministic replay tools implemented in user mode typically have a harder time recording these events than tools implemented at a lower abstraction level. For instance, a deterministic replay tool living in the virtual machine monitor will have access to all interrupts that enter the guest operating system and can thus implicitly solve the problem of recording thread scheduling because scheduling is based on the timer interrupt coming from the processor. To the user mode program and the user mode deterministic replay tool these interrupts are invisible; the only time they can be observed is when the address space or the execution changes. However, unless the operating system explicitly sends notifications to user mode about these events they can be very challenging to detect. In this section we assemble the common techniques used to detect and handle non-determinism in user mode.

4.4.1 Transparency

In Section 2.2.2 we described how dynamic binary instrumentation frameworks put a great deal of effort into becoming invisible to the program they are instrumenting, as not doing so could have detrimental effects on the program's execution. In this section we continue this discussion, but in relation to deterministic replay.

To uphold the fidelity of the deterministic replay tool, we need to be able to guarantee that the replay execution follows the recorded execution exactly. If the recorder (or replayer) reside within vicinity of the program

being recorded (e.g., the recorder shares address space with the recordee) this may become problematic, as the recorder's presence may corrupt the recording.

For example, consider a simple scenario where the recorded program allocates some memory. The library memory manager is deterministic and will select a location in memory to return to the program that fits the requested size. We will call this location *X*. After this allocation the deterministic replay tool performs an allocation of its own and receives location *X+1* from the memory manager. Finally, another allocation is performed by the program and the returned location is *X+2*. When the program execution is replayed the exact same steps will be taken by the program through deterministic execution, but since the recording component of the deterministic replay tool is not present any longer there will be a "gap" in the expected sequence of allocation requests. That is to say, when it is the recorder's turn to allocate memory, no such allocation happens because the recorder has been replaced with the replayer, and it is not guaranteed that the replayer will attempt to allocate memory at the same exact time the recorder would have (if at all). The result of this is that the program will receive the locations *X* and *X+1* instead of the original *X* and *X+2*. If a conditional branch further down the road depends on this information to make a decision, there is now a chance that the decision will be different from the one made in the original program execution. In other words, the recording has become ***corrupted***.

This phenomenon is not exclusive to library interaction and may happen as a result of any value in memory being different from what it was in the original program run. It is therefore crucial to correctly record all address space interaction, including the tiny nuances that may not appear important at first glance. This is *very* challenging to do with complete certainty and therefore no real guarantees exist that the replay execution will mirror that of the original execution. The best we can do is try to lower the risk of this happening as much as possible, and implement safety measures to detect when an erroneous decision was made during replay so we can abort the replay (continuing past the point of a corrupted branch dramatically increases the chance of further corruption as every decision thereafter will be made based on invalid information – the risk of failure increases exponentially). Another downside of having the deterministic replay tool in close vicinity of the recordee is that it is vulnerable to detection by malware.

Deterministic replay tools implemented at a lower abstraction level typically do not have this problem, because their presence is not visible to the

recorded program. For tools restricted to recording in user mode this is a real problem, however. rr [82] solves this problem (or at least reduces the risk) by recording and replaying from a separate process [3], thereby avoiding perturbation of the recorded program. Another advantage of doing this is that a faulty program execution could end up corrupting the memory allocated for the recorder (if both share the same address space), and by running in a separate process we are not exposed to this risk [3].

The reason some deterministic replay tools sometimes reside within the process address space in the first place is because it offers lower performance overhead. Context switches between processes can be expensive, and by remaining in-process we can record with less slowdown than if we had to context switch to a separate process whenever an event needed to be recorded. The question is, however, is the speedup worth the risk? rr reports a 100x speedup by selectively handling some system calls inside the process itself instead of context switching to the recorder process [91], so one could argue it may be worth it.

4.4.2 Recording Non-Deterministic Instructions

When recording in user mode without instrumentation it can be challenging to detect when a source of non-determinism is encountered. *Non-deterministic instructions* is a good example of this. A non-deterministic instruction is an instruction that may give different results on every execution. This is not a problem if we are able to detect and record the result of the instruction, but this is not always the case. For instance, the x86 architecture has a few non-deterministic instructions, but no way to let the program know if they were encountered without operating system support. We will now look at the most common such instructions on x86.

rdtsc

By far the most common non-deterministic instruction used is the *rdtsc* (ReaD TimeStamp Counter) instruction. rdtsc reads the processor's current instruction count since the last reset, and as such can be considered as a source of time to the system. The instruction returns a 64-bit timestamp value in EDX:EAX, and we therefore need to log those registers. The problem with rdtsc is that it is a two byte instruction, so it is not possible to overwrite it with a jump or call (that is, hooking it), because a long

jump instruction is five bytes, and so is the call instruction. A short jump (two bytes) could be possible, but the jump offset can reach no further than approximately 127 in either direction, and chances are very slim that there exists a hole within that range where another jump could be placed without overwriting other instructions.

rr [82], a deterministic replay tool which does not use instrumentation, manages to record the rdtsc instruction despite this limitation. rr uses the *prctl* API in Linux to make rdtsc trap and cause a SIGSEGV signal to be issued to the program. From there they simply check what instruction issued the signal, and if it was rdtsc they record the return value and emulate it [91].

The underlying mechanism that makes this possible is the *TimeStamp Disable* bit of the CR4 control register. When the TSD bit is clear the processor's timestamp instructions (there are multiple) can be invoked in user mode, but if the bit is set they can only be invoked in kernel mode. When a user mode program attempts to execute a rdtsc instruction with the TSD bit set, the operating system will receive an exception from the processor. The operating system can then notify the user program if configured to do so by an API such as prctl. ReVirt [25], a virtual machine deterministic replay tool, also uses the CR4 TSD bit to trap on rdtsc, so this technique is not limited to user mode deterministic replay tools.

In Windows there exist no APIs to make the processor's timestamp instructions trap, and it is therefore not possible to record rdtsc this way without using a kernel driver. However, Windows instead offers an API, *QueryPerformanceCounter*, that replaces the use of the rdtsc instruction altogether. QueryPerformanceCounter returns the current value of the performance counter (not to be confused with the timestamp counter, Windows uses another term for generality). Microsoft recommends using this API instead of the primitive rdtsc instruction because Windows may decide at boot time to use another source than the processor's timestamp counter for its own high-precision performance counter (due to accuracy and portability reasons) [66]. This is good news for deterministic replay tools trying to record rdtsc on Windows, because executables compiled for Windows typically use QueryPerformanceCounter instead of issuing rdtsc directly, and by hooking QueryPerformanceCounter it is possible to record this non-deterministic event.

cpuid

The other common non-deterministic instruction worth mentioning is *cpuid*. cpuid is an x86 instruction that allows software to query the hardware about the available features of the processor. Windows offers an API, *IsProcessorFeaturePresent*, that appears to give access to a subset of the information stored in cpuid, and since it is a public API it can be hooked and recorded like QueryPerformanceCounter. For native use of cpuid, however, there is not yet an easy way to trap to the operating system as with rdtsc (according to [82] modern Intel processors do support trapping cpuid but operating system support is currently lacking). It would be valuable to be able to trap on cpuid and mask out certain information from the result to safely disable other non-deterministic instructions such as *rdrand* and hardware transactions (*xbegin*, *xend* etc.) [82]. This way we would be able to limit the surface of non-deterministic instructions available to the program.

4.4.3 Recording Thread Scheduling

The threads in a multi-threaded program can communicate by reading and writing to the program's address space. The order of reads and writes made by different threads is a source of non-determinism determined by the operating system scheduler. Data races occur when the order of memory accesses was not expected by the developer, and to correctly reproduce these races the exact ordering of shared memory must be recorded. Russinovich and Cogswell recognized that to reconstruct this ordering it is not necessary to record all the shared memory accesses themselves, but rather the thread scheduling [95]. This assumes a single-processor machine is being used, or that all the threads run on the same processor.

Recording the thread scheduling is not easy, because from the perspective of the user mode program it is not obvious when a thread switch happens since it may occur at any instruction. Some early deterministic replay tools (e.g., [98], [100]) did not support multi-threaded recording for this reason. It is possible to get notified by the operating system (if supported) when thread switches occur by debugging the program, but this requires a separate process to act as a debugger for the program being recorded. This comes with performance implications, because the operating system must context switch the debuggee over to the debugger for every single thread switch. This is the approach taken by rr [82]. rr uses *ptrace* to control the program that is

being recorded and is notified of all its thread switches. rr then gets to decide whether or not the switch is allowed to happen and essentially controls the overall thread scheduling of the program ([82], [91], [3]).

Scheduling can also be performed manually within the program itself (*user mode scheduling*). This was demonstrated by Russinovich and Cogswell in their paper where they present their *repeatable scheduling algorithm* [96]. Shortly put, they use a thread with higher priority than the others to control the thread orderings by letting the thread act as a scheduler. The scheduler thread only allows a single thread to run at a time.

A different approach was demonstrated by Geels [30] that uses a mutex to control scheduling. Every thread waits to acquire the mutex, and the thread currently holding the mutex surrenders it at predefined library boundaries. The recorder previously instrumented these libraries to add logging functionality, and as threads pass through they simulate user mode scheduling by passing over the mutex to the next thread which then logs the thread switch.

Windows offers a framework for user mode scheduling [76], but since it is part of the Windows API it must be compiled in with the program, something which clearly violates our design requirement of recording unmodified programs.

Russinovich and Cogswell later extended their proposal to include operating system modifications [95]. They suggested that by adding a few system calls it would be possible to configure the operating system to notify the program of thread switches, similar to that of notifying a debugger. A similar technique was shown in iDNA [8], where the deterministic replay tool can either be notified with the help of operating system modifications, or by using a kernel driver.

4.4.4 Replaying Thread Scheduling

Recording the thread scheduling is only half the problem; the other half is correctly replaying it. The major problem with recording a thread switch is that it may occur on any instruction without notice. Interestingly, this is the same problem we face when we attempt to replay the thread switch, but in reverse, because we need a way to stop at the exact instruction where the thread switch occurred in the original program execution. Once at the location of the recorded thread switch we re-simulate it with the help of the replay thread scheduler.

The way to solve this problem is to count the number of instructions executed up until a thread switch happens and then reset the counter. When replaying, the counter is initialized as the final value and decremented until it hits zero. At that point the thread switch is re-simulated. That is the general idea; the different implementations may decide to log the values differently. The instruction counting happens during the recording phase and must be implemented in addition to the previously mentioned techniques to record the thread scheduling.

Counting Instructions in Software

Static Instrumentation It is possible to include an instruction counter at compile-time into the program. An example is the implementation presented by Mellor-Crummey and LeBlanc in [63] which has less than 10% overhead. Their idea is to use a register or memory location to hold the number of counted instructions, and every instruction is prefaced with an instruction that increments this value. The obvious drawback is that this increases the code size and the program's execution time by at least a factor of two.

A better way is to only count basic blocks and let the instruction pointer precisely determine the exact instruction being pointed at within the basic block, or count the number of backward branches executed. By modifying the compiler we can take advantage of its knowledge of the program structure and optimize the points at which to instrument. We can also tell the compiler to reserve a register to hold the counter and not use it for other operations.

Unfortunately it is not enough to only instrument the program itself, because then the instruction counter will not be accurate enough. For instance, if the execution was to enter a library routine, how would we know how many instructions had been execution when it returned? What if a thread switch occurred while inside the library? To solve this the libraries must also be instrumented. However, the authors recognize that it would take a considerable effort to augment a production compiler to support instruction counting in this way. They therefore direct the compiler to output assembly code (a feature which is supported by modern compilers) and instruments this code instead.

The instrumentation also includes code to support replay. Instead of counting instructions incrementally the counter is initialized with the final value and decremented. When the counter reaches zero a handler routine is called.

A deterministic replay tool that makes use of the presented software instruction counter is the system created by Russinovich and Cogswell [96]. They later changed their system to use hardware instruction counters instead.

Dynamic Instrumentation Statically adding the instruction counter into the program and its associated libraries is very inflexible with regards to recording unmodified programs. Fortunately modern tools exist that allow us to perform this instrumentation dynamically using binary translation (e.g., Pin [59]).

To use the same instruction counting technique when replaying we must recreate the address space from the recorded logs. Afterwards we need to direct the instrumentation tool to execute from the same location and with the same register context as seen in the original program run. Execution will continue deterministically from this location, just like in the original recorded program. The instructions can then be counted using standard dynamic binary translation. An example of a deterministic replay tool doing this is BugNet's replay component [80]. It uses Pin's *ExecuteAt* API to continue execution from a specified location after the address space has been recreated.

Counting Instructions in Hardware

Modern processors possess *hardware performance counters*. Performance counters make it possible to profile the execution of programs. Deterministic replay tools can use hardware performance counters as instruction counters (as seen in e.g., [43], [25], [82], [10], [95]). Instruction counters in hardware have the advantage that they are faster than software counters and less visible (i.e., better transparency). The only overhead they incur is when they are configured or read [3]. They allow us to quickly replay the program without needing to instrument anything because we can just configure the hardware to interrupt us at the target instruction. Hardware counters also do not need to be built into the program, making them the more flexible choice [95].

Complications On the Intel architecture there are two hardware performance counters that are of specific interest to deterministic replay: the *instructions retired* (INST_RETIRED.ALL) counter and the *branches retired* (BR_INST_RETIRED.ALL_BRANCHES) counter [10]. The branches retired

counter counts executed branches and the instructions retired counter counts instructions executed. When counting instructions it is not sufficient to only rely on the instruction pointer, since it may have the same value for multiple iterations in a loop. Together with the branch counter it becomes possible to distinguish the correct instruction from the rest [10]. However, for string operations that are repeating there is an additional level of looping that needs to be taken care of, therefore also the ECX register must be saved together with the two other values (ECX holds the number of remaining loop iterations of the string operation) ([10], [25]).

One could wonder why the counter for retired instructions is not used for this purpose instead as it appears to be the more fitting counter. The answer is that it is not accurate enough for our needs. For example, an instruction causing a page fault would increment the counter and then load the virtual memory page, then re-run the instruction, yielding a count of two instead of one. Hardware counters that suffer from this problem are referred to as *non-deterministic counters*, because the counters emit different results on identical executions. The branch counter on the other hand does not suffer from this problem and can therefore be used to accurately count the number of executed instructions in the form of branches executed [82].

However, the branch counter is still not perfect, because it can be delayed. If the *performance monitoring unit* is set up to produce an interrupt after a certain amount of events, the interrupt actually happens after an additional unknown interval, because there is jitter. To resolve this problem it is necessary to configure the counter to raise the interrupt at an earlier point ([3], [10]). The question of how early this should be depends on the implementation (e.g., rr uses a value of 55 [3] and XenTT [10] a value of 120 by default).

Usage When a thread switch occurs during recording, the retired branches counter value is logged. At replay time the same counter is configured to raise an interrupt when it expires after the recorded amount of branches have been encountered. When the interrupt occurs, it is possible the execution halted in between two branches due to the delay described above. To reach the target instruction the deterministic replay tool must either put a breakpoint on the instruction, or single-step until the instruction is hit. The correct instruction to stop at is determined by contextual information recorded in the recording phase (e.g., instruction pointer, register contents, and so on)

([3, 25]). Finally the thread switch is performed (or another event is injected; the technique of instruction counting can also be employed to determine the location of other recorded events, not just thread switches).

Downsides There are a few downsides to using hardware performance counters for deterministic replay. Hardware performance counters are not available on all platforms (or even moderately modern processors), and if they are they may not be suitable for deterministic replay. For instance, Wu states in his thesis that this is the case for current ARM processors [110]. Another downside is that the processor used for recording must also be used for replaying since the hardware performance counters may act differently on different processors (assuming the same counter is available on both processors in the first place) [80]. This can become problematic if a user wishes to submit a bug report in the form of a program recording to the developers when they have different processors. Last to mention is that access to hardware performance counters requires the processor to be in kernel mode. On Linux the performance counters can be accessed through the *perf* API, which is what rr leverages [91].

Hardware Performance Counters on Windows On Windows there is limited access to hardware performance counters.

Windows offers a performance monitoring API, but the performance counters that are available only answer questions such as the amount of disk-, memory- and network performance of the program. In essence, it is an API made for improving Windows programs' overall resource usage, and it is insufficient for deterministic replay.

There is another API, *Hardware Counter Profiling* (HCP), that is more in line with what we are looking for. It lets the program ask questions about thread cycle time and the number of context switches made, among other things. Related functionality that is not part of HCP can also be found in the WinAPI, e.g., *QueryThreadCycleTime*. The HCP API allows the program to query hardware performance counters. However, there is one catch. The API for configuring which hardware performance counters to use is *KeSetHardwareCounterConfiguration*. Windows functions that start their names with Ke or Ki are kernel functions, that is to say you need a driver to use them [71]. Additionally, the *Performance-Monitoring Counter Enable* bit of the CR4 control register is set to zero by Windows which means that

the *rdpmc* (ReaD Performance-Monitoring Counters) instruction can only be executed in kernel mode, otherwise a privileged instruction exception is thrown by the processor. It is therefore not possible to read the hardware performance counters at all from user mode without using the HCP API, but to select which hardware performance counters to query a driver is required. It is therefore not possible to use hardware performance counters without operating system support on Windows.

4.4.5 Recording System Calls

System calls can be recorded relatively easy from user mode, but despite this, there are some challenges.

System Call Semantics

Most deterministic replay tools that record system calls directly (that is, from user mode; recorders at a lower abstraction layer may record the system calls implicitly by recording other non-deterministic events) depend on a whitelist. System calls not in the list are not supported, because the deterministic replay tool did not yet implement a way to handle those system calls. When such a system call is encountered, the deterministic replay tool either exits gracefully or the program crashes (or worse, the system call is not recorded properly but the execution continues regardless).

On some operating systems it may be very challenging to correctly handle system calls, because the semantics of each system call is not obvious. For example, Windows system calls are for the most part not documented (on purpose) and may change at any time (but are unlikely to; the same syscalls have been used since the early Windows days with minor adjustments). It is not clear what parameters a system call takes or what it returns, nor its side effects on the running program. Some system calls are documented, but the majority remain undocumented. The most effective way to decipher the operation of a system call is to find its equivalent WinAPI library function and follow it down to where the system call is made by looking at the disassembly, because many of the APIs just act as a thin wrapper that prepares the parameters for the system call. Additionally, if an error was returned by the system call, the function translates the error into a more user friendly version that can be inspected with other WinAPI functions.

Linux on the other hand is the exact opposite from Windows; the system calls on Linux are extensively documented, and if all hope is lost, the source code is public. This makes Linux an attractive operating system to implement deterministic replay tools on, because there is a great deal of control available to the developer. As an example, rr supports 129 system calls, which is quite a large number [3].

Uniform Recording of System Calls

It would be very convenient if there existed a mechanism which could detect the changes made by a system call so that no system calls had to be explicitly handled. This is possible in theory by inspecting the changes that happened in memory before and after the system call, but the problem is that this kind of mechanism is not readily available, especially not without operating system support.

The most straightforward way to achieve this would be to clear the *dirty bit* of all the user space entries in the process's page table before the system call is processed by the operating system, and then report which page table entries had their dirty bit set (page tables are described in Section 2.1.3). The page table entries that reported having been modified during the system call would have their corresponding pages recorded as part of a checkpoint. The checkpoint would then be applied to the address space at replay time after the system call returned. Since a system call's return value and side effects are only reflected in memory it should be possible to replay the system call by only recording the memory changes (and the processor's registers which are visible in user mode).

Because we cannot know which bytes actually changed within a page without doing a comparison we are forced to record the whole page. To compare we would need to store every page from before the system call was made, since there is no way for us to know what pages are going to need to be compared against after the system call. This would incur significant overhead, not only the time taken to store the pages but more importantly the fact that the whole address space would have to be stored. Programs using more than half of the available address space would have to store the remainder on the disk, which could prove to have fatal performance consequences, because it would not fit in RAM. The downside of scanning the virtual memory for changes is that it requires operating system support (page table interaction requires kernel mode). The alternative would be to

query the operating system about which user space allocations exist, and then compare or checksum the whole address space, which is magnitudes slower.

IGOR [27] does something very similar to the mechanism just described. Instead of scanning the memory for changes after a system call it does so at intervals, but the result is more or less the same. IGOR uses a modified operating system, however, and queries for memory changes through a custom system call from user mode. Tornado [17] is another contender; its authors recognized that only a few write primitives are used to write into user space from the Linux kernel, therefore by instrumenting those few points of entry it is possible to detect which parts of the address space were modified. Tornado can then invoke ptrace from user mode to query for changes and see what needs to be recorded. Because the instrumentation happens in kernel space this implementation also relies on modifying the operating system.

Chapter 5

Dragoon: A Deterministic Replay Tool Design Proposal for Windows

Throughout the previous chapters we have explored the common categories of deterministic replay tools and how they compare against each other. We have also discussed techniques for recording and replaying, with extra focus on doing so in user mode. For almost every single deterministic replay tool presented there has been one problem in particular that is easily overlooked. Most deterministic replay tools are implemented on Linux and very few on Windows. This is especially true for deterministic replay tools in pure user mode, of which there are virtually none on Windows.

Deterministic replay tools that feature Windows are not necessarily implemented for the Windows platform. For example, virtual machine deterministic replay tools may allow recording of Windows programs, but that is just because Windows is the guest operating system. These recorders may not offer easy deployability since they require set up, or may pose other challenges (e.g., VMware Workstation's record and replay feature was discontinued [93], and PANDA adds significant overhead due to using software emulation [24], something which is not suitable for highly performant programs). There is therefore not much to choose from for the average user.

To facilitate more options, the next step would be to implement a deterministic replay tool in pure user mode in hope of achieving better deployability and speed, but as witnessed in Chapter 4 this is not an easy task due to the limitations of user mode (especially pure user mode). Much of the

previous research has focused on user mode implementations, but done so on Linux. Due to the differences between Linux and Windows, much of this research is therefore, even if successful, insufficient to get the same results on Windows. Additionally, the majority of this research relies on operating system support, further limiting its usefulness with regards to Windows.

With this in mind, we asked ourselves if it was even possible to implement a deterministic replay tool for Windows at all. The answer to this is yes, because some deterministic replay tools have already been implemented for Windows. Expanding upon that question, we instead asked if it would be possible to implement a *high-performing, flexible, generic and publicly accessible* deterministic replay tool for Windows. Previous tools have relied on instrumentation (e.g., iDNA [8]) and thus suffer from performance issues. Others require program modifications at compile-time, but then they cannot be used on unmodified programs. For example, R2 [33] lets the developer annotate the program so that the deterministic replay tool knows how to handle inputs and outputs to the program. The authors manually annotated 1300 WinAPI functions to aid with this on the library side, and additionally did the same to MPI's and SQLite's public interfaces. An approach like this suffers from high implementation cost and inflexibility with regards to unmodified programs.

In this chapter, we propose the design of a deterministic replay tool made specifically for Windows. The design actively avoids the aforementioned problems by taking an entirely different approach to recording and replaying. Instead of using instrumentation or program-, library- or operating system modifications we opted for the classical paradigm of using determinism for recording and replaying. That is to say, instead of spending time and resources on inspecting and recording the program execution we instead identify the sources of non-determinism and only record those. Then at replay time we let the program deterministically execute until a non-deterministic event has to be reproduced to allow the execution to continue. This idea is not new and most of the previously mentioned deterministic replay tools employ a technique similar to this, but in Windows's case there are more restrictions that make it challenging to implement such a design in practice, and that is what this thesis aims to look at.

Throughout the rest of this chapter every reference to program execution will refer to 32-bit Portable Executable native programs compiled for Windows. Likewise, every reference to the operating system will refer to Windows 7 64-bit on the x86 architecture. The execution environment in

which programs are recorded is WOW64.

The code for the project can be found in Appendix A.

5.1 Design Requirements and Limitations

In this section, we will present the design requirements of our proposed deterministic replay tool, Dragoon. These were decided upon before the design process started to guide our implementation in the right direction, and to know the system's limitations from the beginning to avoid unnecessary problems along the way.

5.1.1 Availability

First and foremost, Dragoon was created to fill the gap in the deterministic replay world on Windows. As previously mentioned, iDNA [8], which is Microsoft's deterministic replay tool and undoubtly one of the best such tools on Windows, is not publicly available to the average user. No other deterministic replay tools for Windows appear to be easily available, either. This excludes tracing tools, because we consider those to be different from deterministic replay tools in that they cannot replay a program recording and mostly just log plain data (see Section 3.2). However, depending on our needs such a tool may suffice, and many exist that perform common tasks such as logging at the system call interface (so-called *API monitoring tools*). We again emphasize that there is no replay component involved; the two classes of tools are very different in that regard.

To facilitate easy deployment, we cannot rely on modifications to the program, the system libraries or the operating system. This excludes the use of any kind of kernel drivers. Doing so would severely limit the genericness of the tool for potential non-Windows operating systems, and also increase the set up cost for the average user. Dragoon is mainly targeted at reverse engineering of binary executables, and therefore it is clearly not viable to only allow the tool to work on programs where we have control over the program's development environment. Dragoon should therefore work out of the box on any 32-bit Portable Executable program.

Additionally, Dragoon should be able to faithfully record and replay multi-threaded programs, because most reasonably sized programs will be multi-threaded.

5.1.2 Flexibility

Dragoon is not meant to assist with typical tasks such as debugging crashes or finding data races. There exist many tools that do this already (at least on Linux). Instead of risking our design becoming too domain-specific by attempting to include the necessary functionality to enable the above use cases, we instead opted for a generic approach. Dragoon is designed for reverse engineering tasks, but this does not disqualify it from performing other tasks. The most important use case for deterministic replay is debugging, and this is a large part of reverse engineering, therefore Dragoon should implicitly cover most areas related to debugging.

The infrastructure of Dragoon should facilitate time-travel debugging and automated analysis (in the context of reverse engineering). It should be possible to treat Dragoon as a black box that responds to queries. This component, from now on referred to as the *replayer*, should be an independent component implemented on any operating system. Even though the recording phase happens on a Windows system, that does not mean that the replay phase must as well. The recorded logs outputted from Dragoon's recorder component should be in a compressed binary format that can be shared and interpreted by another replay component. The difference between such a log produced by Dragoon and a generic tracing tool is that the data will not be human readable as it is meant to be replayed and not parsed by scripts.

The only natural requirement of the replay component is that it is somehow able to recreate the address space from the logs and emulate x86 instructions deterministically within that address space. This should be possible by using one of the many x86 software emulators that currently exist. The advantage of separating the recording component from the replay component is that we can make use of the vast amount of tools available on other platforms, e.g., Linux. Data can then be extracted from the recorded logs by quering the replayer in a platform-independent manner, and then handing the returned data to other analysis tools.

By interfacing with Dragoon's replayer, developers should be able to implement scripts for automatic analysis or plug-ins that they integrate into their favorite debugger to manually time-travel (most existing deterministic replay tools have only made their time-traveling features available in gdb or WinDbg). The replayer, as a black box, should take the burden of dealing with low-level details away from the developer and move it into the replayer itself. By encapsulating the details, the replayer will be able to make use

of optimizations internally to improve its overall performance, because the replayer will know how to best utilize the log format. Dragoon's replayer interface should be very low level, preferably written in C, to allow other higher-level language bindings to be implemented. This will ultimately depend on whoever implements the replay component, however.

It should be possible to revisit every instruction and every memory access since the beginning of the recorded program run, therefore the recording granularity must be that of instruction level- and byte granularity. The scriptable API that makes time-travel debugging and automated analysis possible should offer functionality such as (inspiration drawn from [8] and [24]):

- Executing forwards and backwards: Either a predetermined number of steps or until a breakpoint is hit.
- Go to an arbitrary point in time (a *position*) in the execution history and recreate its state.
- Setting breakpoints:
  - On execution addresses.
  - On memory addresses (read or write).
  - On arbitrary positions in the execution history.
  - On events (e.g., exceptions, module loads).
- Removing breakpoints.

With a generic API like this it becomes possible to write more fine-grained APIs on top that can abstract away details and empower the end-user. For example, features such as searching for strings or byte sequences with wildcards could be implemented. It is, however, up to each individual replay component what this generic API should look like (Dragoon's recorder only outputs a raw log), but for a flexible and powerful time-traveling experience our proposed features should be considered.

Lastly, Dragoon should offer the ability to record a full program run but also custom time intervals specified by the user to facilitate a start- and stop mechanism. By limiting the recording length, the log sizes will be smaller and the performance will be better when not recording, but this comes with the drawback that it will not be possible to explore the program's past state before the beginning of the recording.

5.1.3 Performance

For Dragoon to be usable with highly interactive programs like GUI programs or even games, it is important to minimize its recording overhead. Dragoon is not intended to be used on heavy computing software that can be left to itself to perform calculations, or toy programs (e.g., Capture The Flag). One of the main problems with reverse engineering at the binary level is that there is such a vast amount of unannotated information, and the programs can be quite big, so performing static analysis may be costly and time-consuming. It would therefore be preferable to record the program execution and perform dynamic analysis instead, but for programs with outside interaction the slowdown caused by recording may significantly affect the program execution (it has a high probe effect).

For example, a server communicating with the program may time out and close the connection if the program is too slow to respond due to recording overhead. Another example is the result of a decision based on an abnormally delayed input. To be more precise, imagine recording a first-person game. You tilt the view of the player, but with every mouse movement the game freezes up as the movement is recorded. When you regain control, your hand will not be able to continue moving in the original trajectory in the same smooth manner, and the resulting calculations made by the game will vary from what you intended to record. This is a very contrived example, but it should help point out the detrimental effects imposed by even a small amount of recording overhead.

This is not as problematic for programs with little to no outside interaction, and recording overhead is therefore not as important for those kind of programs (i.e., the difference between 2x and 3x slowdown would not be as significant). The exception is when multiple such programs are to be recorded as a batch; in this case the effect of the slowdown depends on the number of programs in the batch. E.g., if a batch of program test runs would normally take ten minutes to complete, a small slowdown of 6x would increase this to an hour. Even for batches taking seconds to complete this can have a significant effect. Usability research shows that wait times of more than a few seconds cause users to get distracted and frustrated, thus compelling them to use other features or products instead [81]. Common wisdom is that automated program test suites should be as fast as possible for the same reason; if running the tests takes too long the development cycle will be disrupted and discourage developers to test properly.

Due to these reasons we therefore aim for less than 2x performance overhead. This should be low enough to record interactive programs while still being realistic.

5.1.4 Self-Modifying Code

Programs that employ self-modifying code can be challenging to record. Regardless, there exist many such programs, e.g., malware and interpreter engines like JavaScript, therefore Dragoon should be able to record self-modifying code. This does not include non-deterministic instructions emerging as the result of self-modifying code, because non-deterministic instructions are already hard to record in the first place.

5.1.5 Limitations

A typical challenge of recording in user mode is recording non-deterministic events that are the result of external interaction with the program. We will give two examples of such events. The first is arbitrary modifications to the recordee's address space made by other processes (or the operating system). The second is modifications to memory shared with other processes. In short, everything the process itself did not explicitly do to its own address space. Writes to shared memory by other processes cannot be detected without using dynamic code instrumentation or other techniques such as Copy-on-Write. Both techniques are relatively heavyweight.

Dragoon will not record this kind of interaction because it will increase the implementation cost of the design. Address space changes made by the operating system as a side-effect of system calls will be recorded, however. Windows appears to refrain from doing such changes without being explicitly asked to (i.e., if a device signaled an interrupt Windows would not simply update the process's address space right away – it would instead wait for the process to request the updates to be applied through the WinAPI), which is in line with our requirement.

The chosen limitations to simplify Dragoon's design are:

- Inability to record shared memory.
- Inability to record child processes.

- Inability to record special features such as thread pooling and Address Windowing Extensions. However, every API that runs deterministically without using e.g., special user mode callbacks or other custom handling should be possible to record by default.
- Inability to record multi-processor systems. Parallel programs can be recorded, but they will lose their parallelism due to being serialized onto a single processor.
- Inability to record uncommon non-deterministic instructions. Only rdtsc and cpuid will be covered.
- Inability to record 64-bit programs. 32-bit and 64-bit programs are executed in different environments on 64-bit Windows, and supporting 64-bit programs would increase the implementation cost.

5.2 Design Overview

5.2.1 User Space as a Self-Contained Black Box

Every individual execution of a program gets its own instance of an execution environment called a *process*. The process itself is not doing any execution, that is the job of the *threads* within the process. From the perspective of the operating system a process has metadata associated with it. This data describes the process and distinguishes it from the other process instances of the same program. However, in user mode we do not see this metadata. In user mode we see ***memory***.

Every single communication made between the operating system and the process happens through the address space virtual memory. If the operating system modifies the process, this will not be detectable in user mode if the address space is not modified. When the program communicates with the operating system this happens strictly through system calls. The program puts a value into a memory location and references this location with a pointer. The pointer is then put on the stack as an argument or held in a processor register. The system call is then initiated. The operating system responds by either modifying the registers, or by modifying one or multiple memory locations in the address space. For most system calls, the addresses pointed at by the pointer arguments are the ones being updated, because

that is the memory the program wishes the operating system to operate on (e.g., filling a buffer at a specific location in memory). There do, however, exist other cases where the system call side effects are invisible unless one specifically knows what part of memory was updated.

From this, we recognize that programs running in user mode are entirely self-contained. If a program does not perform any system calls, nor receives external stimuli from the operating system (e.g., thread scheduling), everything that happens within the program's process is completely deterministic. A simple program like this could be recorded and replayed very easily by simply taking a memory snapshot of the process address space and later replay from that snapshot deterministically. In the real world, however, no such programs exist, as even an empty program performs a few system calls as part of its load sequence (this is added by the compiler). If such a program *did* exist, the compiler would recognize that nothing really happens that can not be predicted, and the compiler would optimize the whole program away and simply return the final value.

We proceed by identifying the sources of non-determinism affecting the process. To replay faithfully all of these must be recorded. As described above, these all come in the form of either changes to the processor's registers, or changes to the address space. The uniform solution would therefore be to scan the registers and the address space for changes after every system call, record the changes, then apply them at the right time when replaying to achieve the effect of the process being a self-contained black box.

It is worth noting that this is not an original idea; it has been demonstrated to work in practice by other deterministic replay tools, but very few have utilized it. The observation that all process interaction happens through registers and memory only allows us to focus our efforts on recording those changes instead of independently handling every source of non-determinism. This way we can disregard Windows's lack of public documentation, and instead ensure faithful replay, not by relying on said documentation to properly record system calls, but by simply monitoring all changes to the address space and dumbfoundedly apply them during the replay phase. The key point here being that we do not know what these changes entail, nor do we care.

5.2.2 Unrecorded Zones

For speed reasons Dragoon will share the address space of the process being recorded to avoid context switches on every non-deterministic event that needs to be recorded, but this comes with complications. One of them is the concept of an *unrecorded zone*. An unrecorded zone is an area within the process's execution path that cannot be deterministically replayed. Every operation that takes place within an unrecorded zone must be extra careful not to affect the address space outside of the unrecorded zone's designated resources, because the changes will not be recorded and can therefore not be deterministically replayed.

An example is suspending all the threads and taking a memory snapshot, then resuming the threads. If a foreign operation happens in between the snapshotting operation and the resume operations, it will not be possible to deterministically replay from the recorded snapshot with a guarantee of faithfulness, and an unrecorded zone exists. This is because the replay will continue execution from the snapshot, but if a bit was flipped before the threads were resumed, and a branch decision later depends on this bit, the recording will have been corrupted since this bit change was not part of the snapshot. To avoid this problem, we must pay extreme attention to the changes that occur in between the aforementioned operations, and ensure that they are as local and self-contained as possible.

Another example is Dragoon handler functions which contain different implementations in the record- and replay components. The execution will enter such a function, but the results will be different depending on whether Dragoon is recording or replaying. A concrete example is the system call handler set up by Dragoon which intercepts system calls. During the replay phase system calls are not intercepted but instead reproduced from the log, therefore the implementations are different but the interface remains the same. If a thread switch happens during the recording phase while inside the function, what value should Dragoon record for the instruction pointer? The instructions in the function body will not match between the recording component and the replay component, therefore recording the instruction pointer as being inside the function will cause problems. A solution could be to record that the instruction pointer is at the entrance of the function when a thread switch occurs and execution is currently residing within the function body.

Unrecorded zones are clearly very risky and should be identified so they

can be isolated and properly examined to ensure they do not perturb the execution and thereby corrupt the recording. They are, however, necessary for Dragoon to do its job, and can not easily be avoided as long as Dragoon remains an in-process component.

5.2.3 System Components

We will in this section give a brief overview of Dragoon's components.

Recording Component

The recording component records the target program and follows the idea that user space is a black box. It attempts to monitor and record all changes to the recordee's address space. The recordee must be started by the recorder itself, even if recording is not active, due to performance reasons which will be explained below, but this should have little impact on the runtime of the program when not recording. It is possible to record from the program's entry point or an arbitrary location in the execution.

The component takes foothold in the recordee's address space by being injected as a DLL and redirects multiple points of interest. The system call interface is hooked so that all system calls are recorded, and similarly exceptions, APCs and user mode callbacks have their entry points hooked. Dragoon does not employ any form of instrumentation as this would slow down the recording phase.

Thread switches are recorded by serializing all threads to one processor, and thereafter monitoring every switch through what we call a *scheduling garage*. Non-deterministic instructions are also hooked and recorded. Both of these concepts are novel and first introduced in this thesis.

Dragoon uses a custom log format in binary form which is based on building an event stream. Events are recorded to the log and later extracted from the log in sequence. The program's execution history is recorded by performing an initial register- and memory snapshot, from which deterministic execution can follow; it is therefore only the non-deterministic events that must be explicitly recorded.

Post-Processing Component

The logs outputted by Dragoon are in a very coarse-grained format as Dragoon does not take periodic checkpoints (that would lower its performance). Dragoon's philosophy is that fine-grained logs should be created after the fact; focus on performance while recording and produce big logs, then replay and create new logs that are smaller and contain only the necessary data. At replay time there is no pressure to perform as all the non-deterministic events have now been captured, and we can afford to spend the time doing a thorough pruning of the logs without being physically present (to illustrate, we could go get a cup of coffee while waiting).

To achieve performant replay checkpoints must be used, and therefore need to be created (see Section 3.6.2). This can either be done the first time the replay is being executed, or by simply post-processing the logs right after recording finishes. More checkpoints means more space is required, but the result is faster replay, and it is up to the user to decide the size of each checkpoint interval depending on the type of program they intend to replay.

Replay Component

Dragoon's log format has been designed in such a way that it is portable to all operating systems and emulation tools, and therefore not bound to the Windows platform. We implemented a replay component prototype which proved not to work in practice, but which can successfully be used as a post-processor instead. Dragoon therefore does not have a replay component, but all the theory remains the same and can be successfully implemented using one of the other two implementation approaches for a replay component described in Section 5.4.

The replay component reads the initial snapshot from the recorder and recreates the recordee's address space after the fact. All memory and registers are reproduced, and from there it is a simple matter of executing through emulation and all events will be reproduced deterministically. All non-deterministic events are injected at the point in time which they were originally recorded, and the aforementioned Dragoon API can be used to inspect the process state at arbitrary points in the execution history. Because no software or hardware counters are being used the replayer relies on the event stream and the process register context to know when to inject events.

5.2.4 Logging

For portability and flexibility Dragoon uses an abstract event stream interface for its log output. Instead of logging details such as what event caused something to be recorded, it is sufficient to simply say that something was recorded and here is its location, size and value. As an example of this, we look at the initial snapshotting mechanism. Instead of making a snapshot event we simply log multiple *MemoryAllocationAddedEvent* events. There is no indication what the events signify as a group, and the replayer will recreate the memory regions from the log without knowing it is actually recreating the snapshot. Because Windows threads cannot have the ID of zero, we use that ID as the initial snapshot "thread" that is replayed first. Another example is exception events. There is no "exception event" per se, it is simply a *ContextModifiedEvent* for the processor context, because that is what actually happens during the original execution; there is no exception, there is only a change of context (the transition to *KiUserExceptionDispatcher*).

Concurrency

Each thread has its own log file to avoid contention. The replayer switches between log files depending on which thread is currently being replayed. For threads having the same ID the same log will simply be appended to, because from the perspective of the replayer there is no difference.

Event Types

There are two categories of events: *aggregate* and *sub*. Every major event is called an aggregate event and contains multiple sub-events. System calls, exceptions, and thread creation are good examples of aggregate events. When a system call happens there may be multiple changes to memory or the registers, and we need to group those events so it is possible to calculate things such as the total size of an allocation in which multiple sub-regions exist. Sub-events, however, are not related to the aggregate event in which they are held; they are just generic, abstract events. Aggregate events often contain no real data and act more as an indicator to the replayer so that it is possible to implement callbacks.

The aggregate events and their logged data are:

- **SyscallEvent**: System call index.

- **MemoryAllocationAddedEvent**: Allocation size.
- **ThreadStartEvent**: Thread ID.
- **ThreadExitEvent**: Thread ID.
- **ThreadSwitchEvent**: Thread ID switched from, thread ID switched to.
- **ExceptionEvent**: None.

The sub-events and their logged data are:

- **MemoryRegionAddedEvent**: Base address, allocation base address, size, protection, state, data (if any).
- **MemoryAllocationRemovedEvent**: Allocation base address.
- **MemoryModifiedEvent**: Address, size, data.
- **ContextModifiedEvent**: Bitmap of changed registers, CONTEXT structure.

Each event must also describe at what time and location during the execution it occurred in order for the replayer to be able to reproduce the event. This is not necessary for events that occur in known locations such as system calls, as system call events are always emitted from the system call handler and the replayer knows to reproduce them when the execution enters the system call handler during the replay phase. We come back to this topic in Section 5.4.2.

Compression

To reduce log space Dragoon tries to only log data that changed. For instance, the results of a system call do not need to be logged if it is the default value [22]. No instructions executed or memory accesses made are logged as this is reproduced with the help of determinism. The logs are compressed with standard algorithms when the recording has completed (or during as long there are spare cores and log files are split into parts), and if a post-processing component is implemented the final logs will first be pruned to contain only the necessary checkpoints, then compressed with an algorithm like *gzip*. Without a post-processor the log will contain whole memory

pages for every detected page change (since we cannot know what changed), but through post-processing we can do a temporary replay and find the exact values that changed and log only those. Additionally we can implemented optimizations such as e.g., instead of logging an allocation base address which we know is on a 64 kB granularity, we instead divide by 64 kB and log only the *index*, thereby ending up with 2 bytes instead of 4 (an address is in the range 0-0x7FFFFFFF and an index divided by 64 kB is in the range *2 GB (address space) / 64 kB = 32768*).

5.2.5 Transparency

As we discussed in the section about dynamic binary instrumentation (Section 2.2.2), it is important that the recordee does not notice it is being recorded. For an in-process recorder like Dragoon this is particularly difficult to pull off because to do its job Dragoon must share the address space with the recordee. A simple call to a WinAPI function could be all it takes to corrupt the recording in unexpected ways, and only direct system calls are permitted by all Dragoon code, with no exceptions. Additionally, system libraries may hold locks, and if intercepted at a low enough level there may be deadlocks if Dragoon attempts to call another WinAPI function which also needs the lock. These problems extend to third party libraries, which makes it difficult to use pre-compiled libraries. For this reason virtually everything in Dragoon has been implemented from scratch to avoid the aforementioned problems, and this includes both the recorder- and replayer prototypes (the latter also having been implemented in-process, therefore suffering the same consequences). For replayers implemented using software emulation transparency is much less of a problem.

Dynamic Memory Allocation

On Windows, the *malloc* C function and *new/delete* C++ operators end up in the WinAPI to allocate heap memory, therefore corrupting the recording. Unordered maps and unordered sets also do this because they are implemented as hash maps; it is only possible to reserve buckets for them, not memory space, and every new node causes a call to *ntdll!RtlAllocateHeap*. Keys can be pre-allocated and then modified without re-allocation, but then we need to know which keys will be used in advance. To make matters worse, we must be careful not to incur such an allocation request behind the scenes

(e.g., add a node to a map) when intercepting *NtAllocateVirtualMemory*, because that will cause another call to *NtAllocateVirtualMemory* and therefore infinite recursion. The universal solution to most of these problems is to use pre-allocated vectors which are sorted so that we can use binary search (an even better solution is described in the evaluation chapter).

It is difficult to avoid dynamic memory allocation everywhere, therefore Dragoon implements its own memory allocator which uses direct system calls only. The allocator implements splitting and merging in an attempt to avoid performance problems and fragmentation. Regardless of this, the allocator is strictly avoided throughout Dragoon unless necessary, and every data structure is always pre-allocated for speed reasons. Interestingly, the custom allocator itself will be created on the heap and thus corrupts the recording, and we must therefore allocate a designated block with direct system calls to instantiate the allocator away from the heap. To interface with Dragoon's allocator we have overridden the *new/delete* C++ operators, and all calls to *ntdll!RtlAllocateHeap* are therefore routed through to Dragoon's allocator instead. This greatly simplifies the code as we can use dynamic memory as normal. The alternative to a custom allocator would be to pass one into every C++ STL data structure which follows the *std::allocator* interface, but overriding the operators is a much simpler solution.

5.3 Recording Component

5.3.1 Recording System Calls

Dragoon records all system calls by intercepting them at the system call interface which leads to the WOW64 layer (see Section 2.1.9 and 2.1.5). The far jump is overwritten by a hook which redirects all system calls through a handler. The handler upholds the system call calling convention and the interception appears invisible to the program. Arbitrary direct system calls or the deprecated *int 0x2E* system call are not supported.

Threads that enter the handler must be enabled for recording, otherwise the system call will be issued directly with no additional handling. Certain system calls must always be monitored so that Dragoon may modify their arguments, and all threads with the exception of Dragoon's and temporary threads are therefore registered for monitoring. This does not necessarily entail that the system calls get recorded, it just means they get intercepted.

Unregistered threads are not intercepted and as a consequence cannot be recorded either.

Because Dragoon's recording component, which is an injected DLL, is removed from the process's DLL list in the PEB to hide it from the program, Dragoon is unable to receive thread notifications. Instead, Dragoon hooks *NtTerminateThread*, *NtTerminateProcess* and *ntdll!LdrInitializeThunk* to keep track of new threads and threads exiting. Dragoon needs to do one last blocking system call when the last thread exits to ensure no new threads have been scheduled to run from *ntdll!LdrInitializeThunk* (the blocking will context switch to the new thread).

When a system call is being recorded, the first thing to happen is that recording is disabled for the thread. This is to avoid recursion when the handler does its own system calls. Recording is re-enabled when exiting the handler. Next, the system call index is examined. The system call will either be manually handler if such a handler exists, or handled automatically. Multiple threads may be doing system calls at the same time, therefore Dragoon uses an event log per thread to avoid contention.

Manual Handling

Dragoon handles every system call manually to the best of its ability. If we know the semantics of the system call, we can record the arguments and the return values directly with no further ado, including any side effects resulting from the system call. By using manual handling, we obtain a vast boost in performance and it will become clear why that is shortly. Some system calls are difficult to handle manually either because we do not know how they work (Windows system calls are mostly undocumented) or because the results may be difficult to locate. For example, *NtCreateThread* is a system call which creates a thread and thus allocates some memory for the new thread. Since we do not know the locations of these allocations, it is hard to record them.

Automatic Handling

To solve the problems of manually handling a system call, we instead handle them automatically. There are many memory regions in the address space whose purpose we do not know nor what code they belong to, therefore we need a solution which does not rely on our knowledge of Windows internals, but works uniformly with every system call.

In brief, we do this by checksumming the address space before issuing the system call, then once more after the system call completes. We compare the checksums and dump any pages that appear to have been modified. New allocations or allocations that were freed are maintained in a list as an optimization. This style of automatic handling was discussed in Section 4.4.5 and has proven to be very challenging to optimize to fulfill Dragoon's performance requirement.

Care must be taken to avoid detecting changes caused by Dragoon itself. For instance, Dragoon's system call handler uses the thread stack and we therefore need to capture the ESP register as we enter the handler and never checksum what resides below it, because otherwise function calls made by Dragoon will be detected as changes to the stack. Similarly, SEH frames live on the stack and if another SEH handler is installed by Dragoon's system call handler this will update the TEB's *current SEH handler* member. We therefore must ignore this member when checksumming the TEB or the change will be detected even though the program never installed a new SEH handler (the program is waiting for the system call to complete which happens to have been intercepted by Dragoon and could therefore not have installed a SEH handler).

We would like to check the page table for which pages were modified during a system call, but this is not possible without operating system support. There are two ways we can detect address space changes in user mode which will be described next.

Write Watching When allocating memory in Windows it is possible to specify a flag that enables *write watching*. Essentially what this does is that the allocation as a whole can be queried for page modifications in a speedy manner, and is clearly just a wrapper for the page table method discussed in Section 4.4.5. The problem with this approach is that when a program is started Windows allocates everything without write watching enabled. Additionally, images and views (shared memory) do not support write watching at all.

To solve this, Dragoon requires the recordee to be started by Dragoon from the beginning, because then there are fewer allocations in the address space. Although not strictly necessary, it does reduce the time needed to backup all the data and reallocate the allocations with write watching. This also allows us to intercept the *NtAllocateVirtualMemory* system call for every

future allocation so that we can enforce the write watching flag.

This cannot be done for images and mapped views (from now on only referred to as *mapped views*), but what we can do is intercept all system calls pertaining to these and allocate them as private allocations behind the scenes with write watching enabled. Needless to say, this semantically activates the copy-on-write mechanism for these allocations and they cannot be shared by other processes, but that should not be a problem since Dragoon does not support shared memory either way. For this method to work, we must perfectly respond to queries as if the allocation was truly a mapped view, and therefore also need to intercept other system calls such as *NtQueryVirtualMemory*. That is to say, from the viewpoint of the program the allocation is a mapped view and should be able to be treated as such. This is quite important since DLLs are loaded into the address space as mapped views and should therefore respond to WinAPIs expecting mapped views. An additional challenge is that arbitrary system calls may introduce new views into the address space; we need to detect those and reallocate them as private allocations with write watching in that case.

The downside of using write watching is that it is not portable to other operating systems as it relies on an API accessible to user mode. It would be hypocritical to claim this as being pure user mode, but seeing as the API exists it would be unwise not to take advantage of it. Regardless we will now present a proper generic approach.

Checksumming Because write watching is not generic enough to be used on other operating systems, and because mapped views cannot be write watched, we need another alternative. A simple albeit slow approach is to checksum the address space at page granularity before and after the system call, then compare checksums. The difficult part is making this fast enough to be used in practice. A thing to mention about automatically handled system calls is that following a study of system call frequency analysis where we determine and implement manual handling for the most used system calls, the only remaining system calls will be those that are more rarely seen. This means that even though automatic handling ends up having, say, a slowdown of 1000x, this could prove to work in practice if only 5-10% of all system calls are handled this way.

To increase performance, an idea could be to only checksum pages we know for certain have potential to change. For instance, odds are that the

system library code sections (*.text*) will never change, and we could skip those completely.

We will discuss our choices for checksumming algorithms in the evaluation chapter.

Keeping Track of Allocations To keep track of new and freed allocations, and to avoid counting the size of allocations on every system call, Dragoon uses a list to main allocation metadata. The list is updated after every system call and Dragoon intercepts *NtAllocateVirtualMemory* and *NtFreeVirtualMemory* to add or remove from the list in addition to this. By tracking allocations, we can quickly scan the address space for other allocations by skipping past full allocations since we already know their attributes and size, and this lets us focus on new or deleted allocations.

For this to be performant we need to make some assumptions. We assume that no system calls will free an allocation and create a new allocation in the same location, with or without the same size; we expect this to take at least two distinct system calls. We also assume that no system call will commit or decommit memory in an already allocated region; we assume that this may only happen when the allocation is first created or directly through *NtAllocateVirtualMemory/NtFreeVirtualMemory*, i.e., an explicit system call which is possible to intercept.

The list is updated on the following events: *NtAllocateVirtualMemory/NtFreeVirtualMemory* is successful (also system calls dealing with mapped views if having reallocated them as write watched allocations), when a new allocation was detected after a system call, when an allocation was detected to have been freed after a system call, when we start recording we add all current eligible allocations to the list, when we stop recording we clear the list.

We observed that there can be a total of 31768 allocations of 64 kB in a 2 GB address space and can therefore divide the allocation base address by 64 kB to get its *index* in the range 0-32767 (e.g., 0x12340000 / 0x10000 = 0x1234). This allows us to index into static bitmaps to keep track of allocations, but for the metadata we need to use a special kind of bitmap which holds structures instead of bits. Bitmaps allows us instant lookup, and Dragoon uses a homemade variant that outperforms the one found in the standard library. Unfortunately, system allocations do not appear to follow the standard allocation granularity and may be allocated with page granularity. We are therefore forced to use an index that is divided by page

granularity instead, as said allocations may not be allocated on a multiple of the allocation granularity. The alternative would be to specially handle those allocations everywhere due to their non-conforming size, which increases the implementation cost.

To avoid race conditions when updating the list or recording memory changes in general by any thread, we need to introduce an *address space lock*. This lock must be acquired whenever a thread logs a memory event, otherwise another thread may sweep in and incur a data race that modifies the memory currently being logged.

5.3.2 Recording Callbacks

Exceptions

We described how Windows dispatches exceptions in Section 2.1.8. We saw that the *ntdll!KiUserExceptionDispatcher* uses a custom calling convention which places arguments on the stack. We do not know if the dispatcher uses data on the stack that was not part of its signature, so it is not safe to record only its arguments. For this reason we record the whole stack from the point it was before the exception dispatcher was invoked (that is, the "old" ESP) down until the current stack pointer (new ESP). We also record the register context, and that should be all there is to it. Any subsequent exception dispatching and handling should happen deterministically, which also includes vectored exception handling (VEH), which should be recorded automatically. The following *NtContinue* or *NtRaiseException* should be recorded automatically by the system call handler.

Software exceptions raised by the program itself through the *NtRaise-Exception* system call end up in the same dispatcher and are automatically recorded.

The *ntdll!KiRaiseUserExceptionDispatcher*, although rarely used, can also be recorded by hooking the callback. From the disassembly it appears that the stack is not modified during the transition to the dispatcher, and it should therefore be sufficient to only record the register context.

The exception event (like so many other events we would be interested in recording) could be passed to the recorder by the kernel directly by using the *Debugging API*, but this would require a context switch to an external recorder process. Dragoon is an in-process recorder and for performance reasons does not use the debug API. A deterministic replay tool that seems

to be doing this however is the *Windows Deterministic Debugger (WDD)*, which at the moment of writing this thesis, is most likely to be the most recent development of deterministic replay on Windows [34].

Asynchronous Procedure Calls

We described APCs and how the APC dispatcher uses a custom calling convention in Section 2.1.6. To record APCs safely we follow the same procedure as we did when recording exceptions. We record the stack contents from the current ESP up to the old ESP from before the APC dispatcher was invoked in addition to the register context. This should be sufficient to record APC with both three arguments and a single argument.

User Mode Callbacks

In Section 2.1.7 we described how the kernel may transition into user mode to perform some tasks. We record user mode callbacks the same way we record exceptions and APCs. The transition back to kernel mode (*NtCallbackReturn*) should be automatically recorded by the system call handler, and it can be recorded in a way similar to how *NtContinue* is recorded.

5.3.3 Recording Thread Scheduling

As described in Section 4.4.3 it is difficult to record thread scheduling in pure user mode. We have come up with a solution for Dragoon that does not rely on operating system support- or modifications (not pure user mode), using the debugging API to receive thread event notifications (not performant enough if recorder is an external process due to context switches, disallows the process to be debugged by the user if not using an external recorder since there can only be one debugging session per process), user mode scheduling (takes the decision making away from the OS scheduler), or passing over a mutex at library boundaries (modifies the system libraries, deadlocks if the thread enters an infinite loop before passing over the mutex, interferes with the OS scheduler to some degree depending upon whether the mutex spins or yields immediately). We observed that thread switches may happen on any instruction and control will be passed to the next scheduled thread. This means it should be technically impossible to detect when a thread switch occurs in pure user mode.

Scheduling Garage

Our solution is based on the realization that thread switches may be detected by controlling the thread's start location after being scheduled. We came up with the concept of a *scheduling garage* that takes advantage of this observation, which to our knowledge has never previously been mentioned in the literature. We will now describe this concept but must first mention that Dragoon serializes all the program threads to a single processor to avoid data races like some other deterministic replay tools (e.g., [82, 5]), thereby forcing only one thread to run at a time. We do not consider this to be a serious performance problem as modern desktop programs seem to have an average thread-level parallelism of just 2x despite the strides in multi-core technology [3]. Dragoon is not designed to record computationally expensive software.

When recording is enabled Dragoon will suspend all the threads in the process and relocate them to a special function called the *scheduling garage* by changing the threads' EIP register (the WinAPI function *SetThreadContext* can do that). All threads are resumed and one thread will run at a time. The thread announces it is now the currently running thread by setting a variable and then changes its EIP and its other registers to what they were before entering the garage, transitioning its execution to where it was originally. The thread runs until another thread is scheduled and the new thread starts its journey in the garage just like the first one did. The new thread checks which thread is currently running, backs up that thread's register context and sets its EIP register to point back to the beginning of the garage. The new thread then announces it is now the currently running thread and is redirected to its original register context so it can run from where it was originally executing, leaving the garage. This cycle repeats whenever a new thread is scheduled and ensures that all threads stay in the garage when they are not out taking a drive.

Recording thread scheduling is now simply a matter of hooking into the garage at the point where the switch happens and log which thread is being switched to and which thread is being switched from.

Multi-threading

Unfortunately it is not that simple to record thread scheduling as we are now dealing with the typical problems of multi-threading. We need to introduce

locking to ensure no race conditions occur when doing the garage switch. The order of operations happening during the switch is also very important but the details have been omitted for conciseness as the examined scenarios are very technical (as is typical with data race scenarios). Adding support for multi-threading will have a performance impact due to the locking, but our lock implementation avoids spinning and immediately yields the remaining thread time instead. Therefore locking should mostly be a problem when the thread holding the lock is doing a blocking system call.

Event Log Write Lock When the replayer interprets the recording it reads and processes one event as a time. A thread switch is logged to the event log of the thread being switched *from* so that the replayer can change the log file currently being replayed to that of the thread being switched *to*. This means that the next scheduled thread must log the thread switch to the previous thread's log before leaving the garage. What if the previous thread was in the middle of writing something in its own log, like a system call event? To solve this every thread log must have its own *event log write lock*, and the lock must be acquired before writing a transaction of events to the log.

Additionally, if the previous thread happens to be in an unrecorded zone, the thread switch must be logged as being at the beginning of the zone as described in Section 5.2.2.

Garage Lock When control is transferred back to a new thread in the garage the thread needs to acquire the *garage lock* right away. Even though the chance is low that another thread will be scheduled while the new thread makes its way out of the garage, there still exists the possibility of higher-priority threads and the like and we therefore need to lock the garage execution to a single thread until it leaves the garage.

It is extremely important to suspend the previous thread (now descheduled) right away after acquiring the garage lock, otherwise it may continue running without being relocated to the garage (particularly dangerous with high-priority threads). First when safely positioned in the garage may we resume it, though it will remain descheduled for some time.

5.3.4 Recording Non-Deterministic Instructions

We discussed non-deterministic instructions in Section 4.4.2 and shed some light on why it is difficult to record them without operating system support. We will now present what is to our knowledge a novel technique to record non-deterministic instructions in pure user mode with no operating system support.

From the previous discussion we recall that it is not possible to reliably hook a non-deterministic instruction because save for a short jump there is no space to perform a long jump or call instruction, both of which require five bytes. We solve this problem by either manually or automatically overwriting non-deterministic instructions whose instruction bytes span at least two bytes, and instead replace them with software interrupts. By using an interrupt index not used anywhere else we can catch the interrupt through the exception dispatcher, check if the faulting instruction has the specified interrupt index, and if so, execute the non-deterministic instruction manually and record the result, then continue execution where the interrupt was triggered by modifying the context argument passed to the exception dispatcher and updating its EIP (and other registers depending on the non-deterministic instruction) to the next instruction. For non-deterministic instructions that only span one byte (if any) there will not be enough space for a software interrupt except for *int3 (0xCC)*, and it would better to find a privileged instruction which takes one byte and execute that to get the same effect as an interrupt. This technique will not work on non-deterministic instructions that are created as the result of self-modifying code.

This will have an impact on programs checksumming their code, because the checksums will differ. Additionally, caution should be taken so that data that looks like non-deterministic instructions are not replaced. It is preferable to only modify sections containing code for this reason. The alternative to recording non-deterministic instructions would be to abort the replay if one was encountered.

5.3.5 Recording SharedUserData

We have been unable to come up with a good solution how to record the *SharedUserData* structure. It resides in a memory page that is read-only, and is updated at a high frequency, so it is not possible to hook it, guard it, or anything of the sort. The best solution appears to be to identify all the

WinAPI functions that read from the structure and hook those reads and record them. It is not sufficient to record the register context at the function exit, since the read values may have been moved somewhere else in memory and we would therefore have an unrecorded zone. The SharedUserData memory page is hardcoded to address 0x7FFE0000 and should therefore not be hard to scan for statically.

Another idea is to rewrite all reads to a page we control, set a page guard on that page, and then do the actual read when the guard is hit. Most likely this would yield a reduction in performance compared to the first solution.

5.3.6 Recording Self-Modifying Code

Program self-modification happens through executing normal code and does not rely on non-deterministic events. Self-modifying code is therefore recorded by implicitly recording the deterministic execution by logging all the non-deterministic events. Non-deterministic instructions emerging from self-modifying code is an exception to this and is not supported by Dragoon.

5.3.7 Recording GUI, User Input, Graphics and Audio

The topics of GUI, user input, graphics and audio may be well documented publicly but their inner workings are often not well documented. For this reason we have been unable to find out how to best record these kind of events. We will discuss our findings regardless.

GUI Programs

In Windows, GUI programs use a mechanism called a *message queue* to manage the user interface and user input. Every program GUI thread has its own message queue and the operating system will fill the queues with messages (events) pertaining to actions taken in the user interface. The messages are directed to the thread which owns the GUI window receiving the notifications. Typical examples of messages are mouse- and keyboard input in a specific window, moving a window, exiting a window, or re-painting a window.

To retrieve messages there must be at least one *message pump* in the program. This is simply a loop which calls the following functions: *GetMessage* to retrieve the next message in the queue, *TranslateMessage* which

translates virtual-key messages to character messages, and *DispatchMessage* which finds the correct window to send the message to. It is possible for each GUI thread to have its own message pump, but not a requirement. The purpose of the message pump is not just to retrieve messages, but also to signal to the operating system that the thread is now in an idle state. Some messages (called *non-queued messages*) will bypass the message queue and go directly to the window in question (through *KiUserCallbackDispatcher*), therefore Windows needs to know it is safe to do this, and by signaling that the thread is in an idle state Windows gets the green light [90, 83, 12].

When a window was created it could either be a new type of window (*window class*) or use an existing window class. Every new window class must have a *window procedure* associated with it. A window procedure is simply a handler which receives the messages passed through *DispatchMessage* from the message pump, or directly in the event of a non-queued message. Because each message contains metadata about the event the window procedure can further redirect the message to its components or other windows.

To record GUI programs we need to record the system calls made to retrieve queued messages (*GetMessage* and its siblings in the WinAPI). Additionally we need to record *KiUserCallbackdispatcher* to record non-queued messages going directly to the windows.

Console Programs

In console programs without a GUI it appears that input is received through the system call interface, so we record this interface. We also record the user mode callback interface (*KiUserCallbackDispatcher*) in order to capture special events such as CTRL-C and other window and input events, which to our understanding get sent from the *Client/Server Runtime Subsystem (CSRSS)* system process which takes care of handling console programs [109, 105].

Graphics and Audio

Graphics and audio are complex topics that involve different drivers and interfaces. Dragoon records all system calls, user mode callbacks, APC callbacks and exceptions, thereby covering all graphics and audio usage interfacing through these channels. The problem is that graphics and audio tend to work with large amounts of data, and it would not be feasible to copy the

data through different abstraction layers all the time. It is therefore likely that behind the public interfaces there will exist shared memory which is used to communicate directly with the drivers and devices (in a safe way).

Dragoon does not record shared memory, but this may not be an issue as long as the program never reads from shared memory and only writes to it. If a read is made from shared memory, it is impossible for Dragoon to know what was read, but writes made by the program itself are deterministic and thus recorded implicitly. It sounds logical that graphical and audible output to a device would only need to go out, and never in. Our decision not to focus on these areas therefore only limit a small range of programs reading back the results from the devices.

5.4 Replay Component

In this section we describe three ways to implement a replay component. We then discuss key topics of the replay phase, all of which more or less can be applied equally to the different replay component implementations.

5.4.1 Replay Component Implementations

In order to replay a recorded program we need to give the code to be replayed access to the original memory layout which was present during the original program run. The way in which we do this has a determining effect on how the rest of the replay component will be implemented, and it is therefore important to choose the implementation carefully depending on which criteria the replayer needs to fulfill.

Replay Using Software Emulation

Using software emulation to recreate the recorded address space is straight forward. Since the emulator controls every aspect of the execution it is simply a matter of telling the emulator to allocate the required memory. To execute the replay we then initialize the recorded register context and run the code using standard software emulation as described in Section 2.2.3. Non-deterministic events are injected by installing an instruction callback which can halt the execution when the time is right, as described in Section 5.4.2.

Replay Using Dynamic Binary Instrumentation

In Section 4.4.4 we saw how DBI can be used to act as a software instruction counter when replaying. This allows us to stop the execution when it is time for a non-deterministic event to be reproduced from the log. The problem with this approach is that it assumes that instructions were counted in the recording phase using DBI. Dragoon does not do this because recording with instrumentation is too slow, and we therefore cannot know how many instructions were executed in between each non-deterministic event. As previously mentioned hardware counters are also off limits because they require operating system support in Windows (Section 4.4.4). We therefore need to find another way to halt the replay execution in order to inject events.

Non-deterministic events other than system calls have the location at which they occured logged by Dragoon (see Section 5.4.2), and we can use this information to tell the DBI component to halt the execution at those locations. To do this the DBI component must analyze each basic block and determine whether the address of the next event's location is in the block. If it is, the basic block must be exited prematurely by inserting a jump instruction that exits the block. The event is either injected or not injected depending on the specifics in Section 5.4.2 and afterwards the execution must continue from where it was interrupted earlier. The rest of the basic block must therefore also have been analyzed to determine the end of the block. The DBI component used by Dragoon is a handwritten implementation that gives us better control over how the instruction stream is instrumented. DBI frameworks tend to be heavyweight and we are more likely to achieve good performance if we only analyze and instrument exactly what we need.

In order to replay we therefore have to recreate the recorded address space and tell the DBI component to start executing from the register context specified in the recorded log. The execution is redirected whenever there is potential for an event to be injected and no software or hardware counters are therefore required. We note that Dragoon avoids DBI in the recording phase, but this is not necessary in the replay phase as performance is not that much of an issue during replay.

We still have not made any mention of how we would recreate the address space. Recreating the address space without software emulation is not a trivial task and we will now present two ways to do this.

Direct Address Space Recreation *Direct Address Space Recreation* is a way to recreate the address space in its most direct form: by using the native process address space. The exact steps are too involved to fully express here but we will describe a general outline. Dragoon's replayer prototype implements this type of address space recreation and the code can be examined on GitHub [1].

First a new process is created and has its private allocations, images, and mapped views removed. System allocations (TEBs, PEB, SharedUserData) remain as they cannot be removed. The locations of all the recorded allocations are reserved and the replayer's allocations are then made. This is to ensure that the replayer does not steal any of the allocation locations which will later be deterministically reproduced, thus causing a collision. The reservations are then removed and the address space is recreated from the logs. Because the system allocations can not be removed we are unable to recreate those, but we can still overwrite them. We found that it is extremely challenging, if not impossible, to perform this recreation process from within the process itself due to strong dependencies between allocations, and an external process is therefore required.

The DBI component can now execute deterministically using the recreated address space as described above, and memory accesses do not need to be instrumented as the original address space layout is accessible to the program. That is to say that if the program wants to access address X it can go right ahead, because the allocation will have been recreated at its original location.

The replayer must be as transparent as possible to not incur replay corruption; just like the recorder had to be transparent to avoid corrupting the recording. If the address space does not look exactly as it did during the original execution from the perspective of the program being replayed we risk losing the determinism and the program may take another branch and derail the replay. The replayer is therefore forced to use direct system calls and not use any system libraries, essentially the same problem we had when recording. This means third party libraries are out of the question as they are likely to use the WinAPI and not direct system calls, but if they did not it would be an attractive solution performance-wise to remain in-process so that we avoid context switching out to an external process.

Fortunately performance is not as important as it was when recording and

[1] https://github.com/uioatle/dragoon

we can therefore move the replayer into its own external process. This will avoid the transparency issue and the replayer can use third party libraries and the WinAPI. We then set up a shared memory section (for speed) that the replayer and the *replayee process* can communicate through. The replayer process acts as a master and the replayee process as a slave, the former giving commands to the latter. The master does all the DBI work and sends the instrumented block to the slave which then executes it and immediately yields its time slice. To speed up the execution a code cache is being used so that the slave does not have to return unless a premature exit is enforced to inject an event (iDNA [8] reports a 2-6% slowdown once the code cache has been warmed up). It is the last instruction of the basic block that determines whether it should jump to a predefined location and yield or if it should execute more blocks from the code cache, which is essentially the same as doing native execution. The slave then returns to the master which applies the event to the slave process – basically the master manipulate's the slave's address space and the only thing the slave does is execute code bit by bit before asking for more work.

As it appears, if we replay using this technique after having restarted the machine, the replay will crash. At first we believed this had something to do with Windows returning from WOW64 system calls at a predefined address that would have been moved due to ASLR after a reboot. We verified that system calls return on the instruction succeeding the *syscall* instruction issued by WOW64. Therefore we can only assume that the problem lies with one of the three aforementioned user mode callbacks: exceptions, APCs, and kernel-to-user callbacks. After a reboot ASLR moves the ntdll.dll system library to another location than it was prior to the reboot, therefore the location of the library in the recording and in the replay will be different, and Windows will attempt to transition into user mode at ntdll.dll's location, but since the address space has been cleared and recreated, the expected callback code will either not exist or have been replaced with different code. What this means is that it is not possible to replay using the direct address space recreation technique, because it does not permit reboots. However, the technique can still be used for post-processing. When the recording phase has finished we can establish a temporary replay using direct address space recreation, pass through the replay once and prune the logs as described in Section 5.2.3. The code for direct address space recreation has been included in the appendix as, in addition to post-processing potential, we believe no one has previously described how to do it – it has only been mentioned in

passing. Our implementation also does not rely on operating system support, something which is likely to be the case in other research.

The completed version of Dragoon will therefore either use software emulation or the other address space recreation technique we are about to describe.

Indirect Address Space Recreation *Indirect address space recreation* is similar to direct address space recreation, but instead of having the recreation happen in its own process we instead share the address space of the replayer with that of the recording. As was explained above this cannot be done because the allocations are going to overlap and cause collisions. Indirect address space recreation solves this problem by letting the code being replayed access the memory in an *indirect* manner: the instructions that access memory are disassembled and have the memory address redirected through a mapping mechanism. The replayer can then allocate the recorded address space allocations anywhere, because the instructions that attempt to access said memory will have to be rerouted to the allocations' new locations before use.

This is straightforward to do for instructions that use immediate memory operands: simply disassemble the instruction and swap out the address with another. It is not as easy with indirect operands such as *mov [eax+2], 0x123*, because here we need to isolate both EAX and the +2. The resulting output will be multiple instructions that look up the target memory address in the mapping mechanism and replaces the value in EAX with the new address.

For instructions that use implicit operands such as *push* which operates on the thread's stack, it is not enough to disassemble and replace the memory address. Because the stack addresses may be floating around in memory or the registers when the initial memory snapshot was taken it is not possible to use another stack location and simply try to use the same physical stack for both the replayer and the code being replayed (which is quite hard to pull off). Seemingly the only solution is to disassemble the instruction into a primitive intermediate representation, then locate the *load* operation that would transfer the memory address at runtime, and insert some more primitive operations in between that ships the address off to the mapping mechanism. Because additional operations have been added it is not possible to convert them back to the original instruction, and we are therefore forced to use semantically similar instructions that perform the operations of

the primitive ones (e.g., a *load* primitive could be executed as an x86 *mov*). This will have a very significant performance impact as every instruction doing implicit memory accesses would have to be transformed into 10-100 instructions.

Despite the performance hit this technique may still be faster than software emulation, and additionally the problem we experienced with direct address space translation where the replay crashes after reboot is not present using this technique, because the callbacks Windows tries to reach out to have not been relocated. The technique should also be easier to implement than direct address space recreation, and it does not rely on operating system support either. One downside however is that intermediate representation libraries are in short supply on Windows and it is hard work making your own due to the large amount of instructions in the x86 instruction set that would have to be supported.

Previous research (e.g., [80, 8, 82]) has shown that methods similar to direct address space recreation can be used to recreate the recorded address space, but we have never seen it be done with a method like indirect address space recreation and therefore consider it to be a novel idea.

5.4.2 Replaying Events

Determining When Events Should Be Injected

Regardless of the implementation approach taken we should have access to an instruction level callback that allows us to potentially interrupt the execution at any instruction. How this callback is implemented in practice depends on the chosen implementation type.

When an event occurs during the recording phase we must log enough information about where and when the event happened so that it can be replayed. The standard approach is to use instruction- or branch counters, most often in the form of hardware counters, so that we can record the amount of branches executed and the register context at the time of the event. During the replay phase we program the hardware counters to interrupt the execution after the amount of executed branches has been exceeded and from there we may single step until the register context lines up with the one in the recording.

Dragoon does not use any kind of instruction- or branch counters and therefore relies only on register context, which is risky and not always suffi-

cient. Consider the scenario of an infinite loop. The loop may spin a thousand times before a thread switch happens. When we replay we program the instruction callback to halt at the instruction on which the thread switch event occurred, and we then compare the register context to that in the log. We then realize that for the infinite loop every iteration has the same register context, so at which iteration did the thread switch happen? This problem is solved by counting branches, but that is unavailable to Dragoon since we do not use software counters (too slow), hardware counters (requires operating system support), nor instrumentation (too slow). The resulting outcome is that Dragoon can not reliably inject the thread switch event in this scenario. For the purposes of reverse engineering and debugging this scenario and others like it should not be a problem as we are more interested in the execution- and data flow in the recorded program than the exact time at which something happened.

Another example is a loop which calls a function, and on the second loop iteration an instruction within the function raises an exception. The iteration counter will at this point either remain in the register it was assigned to (e.g., if the function does not use that register), or it will have to have been stored somewhere temporarily so that it can be restored into the register when the function exits. By only recording the register context we will not be able to distinguish the two iterations and must therefore also log the thread stack (preferably only a checksum of the data), which has the highest probability of being used as temporary storage.

As non-deterministic events are relatively rare it could be an idea to also checksum and log the whole address space as the performance hit might be acceptable. This would record the instruction counter even if it was saved somewhere other than the stack. Until further experimentation shows that this is warranted we would still recommend against it for performance reasons.

Most other cases should be sufficiently identifiable by register context only, as data stored in memory must be read into a register before it can be used. This is not a chance we are willing to take and we can therefore conclude that Dragoon must record both register context and the thread stack at minimum.

Replaying System Calls

System calls are replayed when the deterministic execution enters the system call interface during the replay phase. The recording component's system call handler has been replaced by the replay component's system call handler, which is technically much simpler. The handler reads the log for the current thread and reproduces the effects of the logged events which occurred when the system call was recorded. A system call event is an aggregate event and may therefore contain multiple sub-events that modify the address space, add or remove allocations, and so on.

Some special system calls, e.g., *NtContinue* and *NtCallbackReturn*, changes the thread's register context. To replay those kind of system calls the target context must be recorded and reapplied on the replay thread; whenever the original thread changed EIP and other context, so must the replay thread. System calls that change the context of other threads, e.g., *SetThreadContext*, must be intercepted by the recorder, otherwise the context change will go unnoticed.

Detecting Mistakes and Checksum Collisions When the system call handler is entered by deterministic execution we must check the requested system call index. If this index does not match the system call index found in the log we can safely assume that something went wrong and the deterministic execution was corrupted by something Dragoon did not record correctly. This should help detecting failing checksumming when recording automatic system call handling; if the checksum claims no memory was modified because there was a collision, then no memory modification events will be logged, and as a result the recording will have been corrupted (we continue this discussion in the evaluation chapter). If the indexes match we still cannot know that nothing went wrong, but a sequence of matching system call indexes should be a reassuring sign everything is working as expected.

Replaying Thread Switches

From the perspective of the replayer a thread switch is simply a change of register context. A *ThreadSwitchEvent* serves two purposes. The first is to inform the user that a thread switch event occurred in case they wanted to be alerted by Dragoon's API. The second is to tell Dragoon to stop reading events from the current thread's event log and start reading the log of the

thread being switched to.

The replayer simulates a thread switch by reading a thread switch event from the log of the currently replayed thread. Because threads were serialized to a single processor we can replay them all by using only one thread. Its ID or settings are not important as long as it executes at the different locations all the original threads did, and in the same order.

The thread switch aggregate event contains multiple pieces of information. It must describe the location at which the switch happened, as explained in Section 5.4.2, the ID of the thread being switched to so the replayer can switch to reading the other log file, and the register context of the thread being switched to. Since all the thread stacks remain as allocations within the recreated address space it is a simple matter of changing the ESP register to pretend to be another thread during the replay phase. The replayer does not swap out the log files before the context change has been applied to the replay thread.

When the replay thread accesses the *FS segment register* we must make sure it returns the correct TEB for the thread being replayed (see Section 2.1.1). Therefore on every thread switch we must also swap out the FS register. With software emulation this is simple, but if we are using any of the other two implementation approaches we must disassemble the FS register access and replace it with instructions that return the correct TEB address.

Replaying Callbacks

Replaying callbacks is very similar to replaying thread switches, and exception-, APC-, and user mode callbacks are all replayed in the same way. We recall from Section 5.3.2 that all the callbacks could be recorded by capturing the register context and the data that was added to the stack by the kernel. It is then a simple matter of applying the register context to the replay thread at the correct location to have it redirected into the recorded callback. The stack data must also be applied to the recreated address space.

5.4.3 Replaying Allocations

Recreating System Allocations

Allocations in the address space are recreated at the location at which they existed in the original process. We therefore need to be able to avoid having existing allocations in these locations, or there will be a clash. The system allocations (TEBs, PEB, SharedUserData) can not be reallocated, but they can be modified (except for the latter). We must therefore spawn threads until the expected TEB locations are taken by the threads, then kill the rest of the threads. It is now possible to overwrite the data of the system allocations with that found in the log without problem.

It should now be obvious why it is very difficult to replay in-process; the replayer will be forced to do direct system calls only, otherwise the TEBs or PEB may be modified by the replayer unintentionally if using the WinAPI, thus corrupting the address space. We can spawn an arbitrary thread for use by the replayer that does not use any of the TEB locations seen in the recording, therefore this is one less thing to worry about. For this reason it is much safer to simply use an external replay process, or if replay performance is not a concern, software emulation.

Recreating Regions and Protections

One would assume that to recreate memory regions we would need to record the attributes of the regions (state, protection, etc.). In order for Dragoon's API to allow the user to query regions about their attributes, this would be true. Dragoon does not do this for performance reasons, thus lacking this ability. Instead, Dragoon relies on the fact that in order for memory to be used by the deterministic replay we do not need to reproduce the attributes of every memory region; we just need the memory to exist, be committed, and be writable.

The Assumed Method Consider the following scenario. The program allocates a memory page which is committed and has the protection *PAGE_READWRITE*. The program moves some data into the page and changes the protection to *PAGE_READONLY*. The program reads from the page and then writes. An exception is raised because the page is read-only. Now consider how this would be replayed. We recreate the allocation, which at minimum must be 64 kB even though it only holds a single page due to Windows's allocation

granularity. We recreate the page and applies the correct protections. We recreate the data held in the page and the program reads the data through replay determinism. The protections are changed and we capture the exception when the page is written to.

From a technical perspective we rely on the page protection to raise the exception. We would therefore have to detect the protection changes of all memory regions throughout the recording phase. During the replay phase we would have to reapply the recorded protections and update them when a change was detected. Additionally we would have to catch the exceptions caused by access violations.

Dragoon's Method There are a few problems with the previous method. First, it is not trivial to capture protection changes if we are to fulfill Dragoon's performance requirement. Secondly, we would not like to have to deal with handling exceptions while replaying; we would like all events to be reproduced from the log, and not by incurring actual kernel interaction during replay. Thirdly, if we are not using software emulation it is difficult or even impossible to recreate mapped views with the proper attributes, e.g., *SEC_IMAGE* for images and we will therefore be unable to recreate the correct attributes for all regions.

To solve all these problems Dragoon instead commits the full allocation, in this case 64 kB even though only one page is being used. The data is moved into the page and the protection is changed in theory, but the allocation remains as *PAGE_READWRITE* and no protection changes are applied in practice. The replayed program writes to the page and there is no exception, because the page is still writable. What happens instead is that during the recording phase, the access violation was recorded as an exception event and Dragoon instead replays this exception event. We now get the same effect and avoid having to spend time recording region protections. Similarly, if the program queries the memory region for its attributes, Dragoon replays the system call and returns the correct attributes even if the actual memory has different attributes. We therefore solve the problem of some region attributes not being recreatable.

Committing the Whole Allocation One question remains: why does Dragoon commit the whole allocation if only part of it was committed during the original program run? Surely this will take up a great deal of space when

replaying? The answer to this is yes. This approach comes with a space cost, but Dragoon is designed for speed, not space. Consider the following scenarios to see why Dragoon must do this.

We recall that thread stacks use a special mechanism to detect when the stack is full (see Section 2.1.3). If a thread pushes something on the stack and touches the guard page the processor will raise an exception which is caught by Windows. Windows moves the guard page down one page and lets the thread keep using the area which was previously guarded – the stack has expanded. The problem here is that we cannot detect when this happens as this is a special mechanism handled behind the scenes by Windows. We can therefore not emit a *StackOverflowEvent* or similar as the stack expansion may happen on any of the push instructions in the program. When we replay and such a stack expansion is meant to happen, what are we supposed to do? The deterministic replay will push a value onto the stack and expect that Windows would take care of the expansion. If we instead commit the full allocation the memory will be available and the stack operation will be successful, because the memory is just waiting to be used with no special handling. If the program was to query the stack's protection Dragoon would simply replay the system call and make it appear as if the protections actually exist. If the program attempts to access the stack that is meant to be inaccessible Dragoon will reproduce the exception even though the memory is accessible in the replay, because in the original program there was an exception.

What would happen if a user mode callback was incurred and the thread stack was about to be full? The kernel places some temporary data on the stack as per the custom calling convention, and this would cause a stack expansion, and we would end up with the same problem which we just described. This problem and others like it are automatically solved by Dragoon's solution to the stack expansion problem.

Lastly we will discuss commits and decommits. If a page was decommited and later accessed, Dragoon will reproduce the exception event. We therefore do not need to record decommits. If a page was committed in a reserved region we would assume that it is necessary to record this, because there will be new data in the committed page that must be logged. This is not true as the system call that commits the page will fill the page with zeros and all subsequent data moved into the page will happen after the system call. By committing the whole allocation during replay the program is free to start filling the page after the system call even though the commit was not

recorded, because the page was already committed. As mentioned previously this assumes that no system call will commit memory within an existing allocation and also fill it with data at the same time, otherwise this approach will not work. This sounds like a reasonable assumption. By ignoring commit- and decommit events we greatly speed up the allocation tracking discussed previously in Section 5.3.1. It is however possible to detect both commits and decommits in a tolerably faster fashion than simply comparing memory allocations before and after the system call, but we will not discuss that here as we have already presented a better solution.

5.4.4 Replaying Self-Modifying Code

It is relatively easy to record self-modifying code, but not as easy to replay it if we are using DBI. Because the instrumentation relies on a code cache to speed up the execution we need to make sure we detect self-modifying code in order to invalidate the target basic blocks in the cache. We must therefore analyze all instructions which perform memory writes and check their target location, for example. This is one of the reasons why we went with our own DBI implementation, because it is not for certain that existing DBI frameworks are able to handle self-modifying code (e.g., Pin was shown to only support a subset of self-modification in [92]) and we did not want to take that risk.

Since we are essentially analyzing every instruction when doing DBI during replay we can detect when a non-deterministic instruction emerges from self-modifying code and abort the replay.

5.4.5 Post-Processing

We mentioned how it could be wise to perform an additional post-processing step after the recording phase in Section 5.2.3 in order to prune the logs as early as possible. Now that we have witnessed how some of the inner workings of the replay component function we revisit this topic and realize there are additional advantages to using a post-processing step. The recording component produces multiple large logs and we would like to prune those logs and merge them into a single log file which is much smaller.

We saw how the replayer has to switch between log files when replaying thread switches (Section 5.4.2). If we post-process the full replay once we can merge all the thread switch events into a single log instead, because

the multiple log files were used in the recording phase as a way to avoid contention.

We also saw how it is important to compare the register context and the thread stack to decide whether the replayer should inject an event or not (Section 5.4.2). By post-processing we can do this heavy lifting once and instead label the time at which the event was injected in the execution history to avoid doing this on every replay run. For instance, instead of doing the aforementioned comparison we could instead transform the log to state that on the *X-th* executed instruction there will be an event.

The downside of having a post-processing step is that it takes time to perform. We believe the wait is worth it as the logs are reduced in size, and other difficult concepts can be made more abstract, as was just described. This should help developers who implement a replay component on other platforms keep the implementation cost low by relying on simpler and more abstract concepts.

Chapter 6

Evaluation

In this chapter we will discuss the observed outcomes of the unfinished Dragoon prototype to get a general feeling of which direction development is headed. The prototype has been under constant improvement in order to fulfill Dragoon's performance requirement, something which perhaps not unexpectedly proved to be very challenging. Deterministic replay tools implemented in software often suffers greatly from performance problems and we chose to focus our efforts on this point in particular to the detriment of Dragoon's other features. The Dragoon prototype therefore remains unfinished and can not be compared to other deterministic replay tools nor be run on real programs at this point in time, in particular because Dragoon has not yet implemented the recording of thread scheduling, and most real programs are multi-threaded. Despite this Dragoon's infrastructure has been prepared to scale to large programs and the groundwork for multi-threading support has been completed.

Because of these reasons we strongly urge the reader to not consider any results in this chapter as final or even realistic; the addition of other features may have significant effect on the current outcome, although unlikely as we have considered the outcomes in theory.

The purpose of this chapter is to showcase the effects of our experiments in order to help us better gain an intuition for how to theoretically design the rest of the system.

6.1 Dummy Program

The dummy program used to test Dragoon's performance is very simple. It is a Visual C++ program that uses a stopwatch to count the time it takes to perform 1000 *NtQueryVirtualMemory* system calls, one of the simplest system calls. The system call is incurred directly, that is to say we do not call *VirtualQuery* in the WinAPI. The system calls have been made to query increasing addresses in the address space to avoid having the same data being returned on every call as this would reduce Dragoon's log output and not be a realistic representation of its performance nor log size usage (Dragoon only logs changes to the address space). The program is made to reset the address if it exceeds a certain threshold, otherwise if the number of iterations is changed to a much higher number the system call may end up querying above the available address space repeatedly, thus receiving the same return data on every system call.

Clearly this dummy program is nowhere near capable of simulating a real program. The dummy is intended to help in debugging Dragoon and this far most of the focus has been in weeding out bugs in the system call handler, therefore we have been interested in the performance of recording system calls. It is important to keep in mind that a real program is going to spend most of its time executing code, not issuing system calls. That is to say, if Dragoon was recording a program which executed 1 trillion instructions but never did any system calls or experienced other non-deterministic events, it would be sufficient to record the initial memory snapshot and the rest of the recording phase would incur a 0x slowdown, because Dragoon would not be recording anything. Knowing this we can conclude that the dummy program is special in that most of what it does is repeatedly issuing system calls instead of executing its own private code with the occasional system call.

Also, even though the dummy program is designed to not return the same data from every system call in order for us to gauge the worst case scenario, real programs are likely to perform system calls that return the same data on repeated executions and thus can take advantage of Dragoon's log-on-change policy, thus reducing log size.

The C++ runtime's startup code must be run before we reach the testing area, and this has been an opportunity to make sure Dragoon can record every arbitrary system call which is included in this process until the test area is reached. Similarly, the code which is run when exiting the program has also been used to make sure Dragoon is able to stop recording and exit

gracefully.

6.2 Memory Usage

Dragoon's focus is almost entirely on performance and memory usage has therefore been neglected. Observed memory usage appears to be around 70-100 MB and is likely to remain rather static. The data structures have been pre-allocated in order to scale and thus use the same amount of memory regardless of whether the recordee is a small or a large program. The alternative would be to re-allocate accordingly, but this will have an effect on performance and more importantly significantly increase the implementation cost.

Some data structures are static and can not be re-allocated, such as bitmaps. In order for Dragoon to achieve $O(1)$ lookup, insertion and removal this was a necessity, and almost all of the data structures used by Dragoon work in constant time and scale regardless of recordee size. Dragoon uses a custom bitmap implementation which is faster than the one offered in the standard library, sacrificing memory instead, and also allows Dragoon to store objects in the map. This means that by default Dragoon must statically allocate memory for all of the objects even if they are not being used. The alternative to using bitmaps would be to use vectors that were pre-allocated as much as possible, and sorted with an algorithm suited to the scenario, then run a binary search on the vector, but this approach is slower.

6.2.1 Conclusion

Dragoon uses the same amount of memory regardless of whether system calls are handled manually or automatically. If we could manually handle every system call we would be able to significantly reduce the memory footprint as we would not have to keep track of allocations throughout the lifetime of the program. This is not feasible and sometimes not even possible in practice because many Windows system calls are undocumented and we can therefore not know for certain how to handle them.

6.3 Log Size

Dragoon does not prioritize log size, but it is an important factor regardless, and we have therefore attempted to reduce the size as much as possible without sacrificing speed. Our initial tests showed that a full execution of the dummy program from start to finish yielded a log size of about 350 MB. For a barebones program that does virtually nothing this was clearly unacceptable. We proceeded to optimize the log size as much as possible and found that the write watching mechanism had a tendency to respond to unexpected changes.

For instance, Dragoon's memory allocator used an internal lock, and whenever the lock was acquired or released a bit would be flipped and the write watching mechanism would pick it up. This was because the object instance had been allocated using dynamic memory, thus corrupting the heap. To avoid this we allocated the custom memory manager on its own designated block using raw system calls.

Similarly, in the automatic system call handler the usage of *NtGetWriteWatch*, the system call which retrieves write watching information about an allocation, seemed to always update some data within the WOW64 layer. We deemed it safe to ignore this region and stopped write watching it, receiving a 10x reduction in log size. In the code this allocation is referred to as *the probed region* (due to how it was always detected to have been probed, but it was difficult to find the source as the return from the WOW64 layer seemingly reverted the modification, thus leaving no tracks), but upon further inspection we suspect that the allocation may be the 64-bit memory heap used by WOW64 in 64-bit WinAPI calls such as *ntdll!RtlAllocateHeap*.

Another important optimization done in the automatic system call handler was to stop write watching the thread stacks and instead temporarily store them and compare them for changes after each system call. Since each thread stack has a maximum size of 1 MB if the whole stack is being used (which is rare), copying the data and comparing it is cheap in terms of speed and memory. The write watching mechanism took note that Dragoon's system call handler lived on the stack, adding call frames there, and would always trigger a full stack dump to the event log for this reason. In addition to this the write watching mechanism also reported multiple other pages in the stack's allocation that needed to be logged on every system call, but we were never able to figure out the cause of this. To solve these problems we captured the current stack pointer (ESP) when entering the system call han-

dler. We then backed up the stack data from the beginning of the stack until the captured stack pointer before issuing the system call and then compared the changes in the same range after the system call completed. With this optimization any changes to the stack can be logged directly as individual *MemoryModifiedEvent* events instead of logging the whole stack. If we used write watching we would not know which bytes had changed and would be forced to log the whole stack. As a result of this optimization the log size was reduced by 1040x.

Another optimization which was mentioned previously is to only log system call return values that were not the default, as was described by the authors of the Arnold deterministic replay tool [22].

6.3.1 Conclusion

A common thread when it comes to log size issues is the automatic system call handler, because it is forced to log modifications at page granularity since we cannot know which bytes changed. With manual system call handling we can know exactly which bytes to log, and the resulting log size should be drastically reduced. *It is therefore important to manually handle as many system calls as possible to obtain full control over the speed and log output of the recording of system calls.*

The current log size outputted by Dragoon for a full run of the dummy program from start to finish is 59 kB without post-process pruning. With WinRAR compression the log size is reduced to 1 kB. This is an improvement compared to the initial size of 350 MB by a factor of 6075 (358400 with compression).

These numbers only account for recorded system calls and we expect a 50-100% size increase by also recording register contexts and stack checksums to tell the location of non-system call events (as described in Section 5.4.2). We expect only thread switch events to have a noticeable effect on log size as they are fairly frequent while the remaining events are rare enough to be of negligible importance.

6.4 Performance

Dragoon's recording component has been implemented as an experimental mix of write watching private allocations and checksumming the rest

of the address space. This is because we considered the approach to emulate mapped views with write watching to have a moderately high implementation cost and be risky. In addition it is not portable to other platforms and we would expect other platforms to have to checksum everything and therefore suffer a drastic performance reduction.

This section discusses performance in the context of automatic system call handling. Most of the focus has been on enumerating the allocations within the address space, detect which allocations can be skipped or write watched, and checksum the rest. We conclude the section by comparing the observed results with the performance of manually handling the system calls instead.

6.4.1 Allocations

We have come to the conclusion that if we use a list to keep track of the allocations as opposed to enumerating the allocations both before and after the system call and comparing them that we gain a performance boost of 13-23%. The new method has shown that it is possible to scan the address space for allocations *only once per system call after the system call has completed*, and it is from this possible to detect commits, decommits, new allocations and freed allocations.

We have found that future work should put more effort into better identifying the system allocations made by Windows, which are undocumented, in order to skip them. For instance, by identifying the anonymous *locales.nls* allocation which contains locale data and skipping it we were able to achieve a 6% speed boost. This is because the allocation must be checksummed whenever a system call is handled automatically, and checksumming is slow.

Allocations that we expect not to change (e.g., the locales allocation, system library .text sections etc.) should be identified and skipped. 64-bit system libraries such as the 64-bit ntdll.dll used by WOW64 should be able to be skipped, and while on the topic we can mention that Dragoon already identifies and skips wow64.dll, but the remaining WOW64 DLLs remain for now and will be taken care of in the future (e.g., the Intel Itanium WOW64 DLLs can most certainly be skipped unless Windows is smart enough to not load them in the first place for processes running under x86).

6.4.2 Checksumming

For allocations that are not write watched, e.g., mapped views, the only option we have left is to checksum all the pages in the allocation and compare checksums after the system call has completed. It is possible to checksum more than a page at a time, but that also means if a change is detected we are forced to log the whole range. It seemed therefore suitable to keep checksums at page granularity.

Checksumming Algorithms

Checksumming effectively proved to be one of the most difficult aspects of implementing Dragoon. We started by implementing CRC32 in software, then transitioned over to using Intel's CRC32 instruction in hardware. This is not a portable approach as we are now locked to the Intel architecture. To avoid this we can use fast well-known software hashing algorithms such as *MetroHash*, *farmhash* and *xxHash* (which claims to reach hashing speeds nearing the possible memory bandwidth).

We compared CRC32 in hardware against xxHash in software and found to our surprise that xxHash outperformed both the 32-bit and 64-bit variant of CRC32. This happened to be due to a programming error as our implementation used a serial approach instead of the parallel approach which is meant to be used with the CRC32 instruction. We noticed this too late to fix it, *but we expect the performance of the CRC32 hardware approach to be tripled by fixing this.*

The current implementation of Dragoon shows the following checksumming performance on average for the dummy program's 1000 system calls using the algorithms below:

- **CRC32 32-bit in hardware (serial implementation)**: 4625 milliseconds.
- **CRC32 64-bit in hardware (serial implementation)**: 4274 milliseconds.
- **xxHash 32-bit in software**: 3504 milliseconds.

You may have noticed that one of the algorithms presented is a 64-bit implementation and wonder how we can use it in a 32-bit process. Windows

does not offer intrinsics for the 64-bit version of the CRC32 hardware instruction unless the program is compiled for x64, so it is not possible to use the 64-bit version of the instruction in Dragoon. Here Dragoon uses a little trick which we already saw in Section 2.1.5 where 32-bit system calls are redirected to the WOW64 layer by switching the execution over to 64-bit by using a special far jump instruction. Because the process we are recording is not a 32-bit process but actually a 64-bit WOW64 process (all processes on 64-bit Windows are 64-bit) we can use this trick to switch the processor into running 64-bit code. We therefore compiled the algorithm for x64 and copied the 64-bit assembly code into Dragoon, wrapped it in handwritten 32-bit assembly code which performed the switch to 64-bit mode and formatted the arguments to fit the new 64-bit environment.

Checksumming in Parallel

For a real program where performance matters, 3504 milliseconds for 1000 simple system calls is clearly unacceptable. Even if we were using the correct parallel CRC32 approach (which would surpass the currently fastest approach) the slowdown would still be quite noticeable.

To ameliorate this we could take advantage of the fact that Dragoon serializes all threads to a single processor. This would leave us with some spare cores, and by using worker threads spread out on those cores we could checksum multiple pages in parallel. This should be possible without contention as every page is checksummed individually from all the other pages even if they are in the same memory region. This should have a strong positive effect on the overall checksumming performance. Additionally we could make it so that pages are checksummed in batch and not one at a time before returning to Dragoon, thereby possibly keeping the CPU cache warmer by keeping the checksumming code in cache for longer.

Checksum Collisions

An important factor to selecting the correct checksumming algorithm is the expected risk of collision with another checksum. Dragoon checksums chunks of 4096 bytes (page size), which is 32768 bits. For a checksumming algorithm which outputs 32 bit, the chance for collision is clearly fairly dangerous. It is very important for Dragoon to be able to detect when a memory page was modified so that it can be logged and maintain the correct determinism

during the replay phase. We realized too late the serious effects this may have on the output and would strongly suggest against using a checksumming approach until a risk analysis has been performed, although the outcome is not likely to make it safe for Dragoon to use checksumming. Additionally this limits the number of available options for non-Windows platforms to have a generic approach to scanning for address space changes without operating system support. This is very unfortunate, but this thesis mainly concerns itself with Windows, and for Windows there exists a solution.

Previously we mentioned how we deemed the use of checksumming to be a safer and less implementation heavy approach than to intercept the system calls of mapped views and reallocate them as private allocations with write watching enabled behind the scenes. In light of this new information we now consider this to be the recommended approach. This solution should also be vastly superior performance wise, likely by multiple orders of magnitude considering how write watching is per allocation and checksumming time grows as more and more pages are allocated.

6.4.3 Manually Handling System Calls

The previous discussion was in the context of system calls handled automatically by Dragoon. By instead handling system calls manually we can skip the address space monitoring altogether and simply record the system calls directly. As a result the performance should see a dramatic increase.

Findings

Our findings are multifaceted. Each event log instance in Dragoon contains a statically sized buffer which is flushed to disk only when the buffer has been filled to avoid doing repeated system calls writing to the file.

We found that it is quite important to tune this buffer size towards the program we intend to record. For example, by using a buffer size of 32 MB, Dragoon records with a 32x slowdown. We increased the number of system calls in the dummy program to 10 million up from 1000 and the performance remained fairly consistent. We then lowered the buffer size and saw the performance double, triple, quadruple, and so on, depending on the size we specified.

The lowest recording slowdown we managed to attain by manually tuning the buffer size was 4.5x for 1000 system calls and 2.2x for 10 million system

calls (unrecorded runtime being 2245.85 milliseconds).

It would probably be a good idea to find a way to automatically tune the buffer size, e.g., by examining the target program before the intended record run, and use machine learning to find the most appropriate buffer size. There seems to be a limit to how far we can take the performance by changing the buffer size, however.

Comparison to Automatic System Call Handling

The total runtime of the dummy program without recording is on average 0.42 milliseconds. When recording the average runtime is 13.84 milliseconds if system calls are handled manually and 3504 milliseconds if system calls are handled automatically. Manual handling therefore outperforms automatic handling by a factor of 253x. We believe that by correcting the current checksumming implementation and parallelizing it on spare cores, or even swapping out the checksumming approach with write watched mapped views altogether, we will be able to drastically decrease this performance gap. For instance, we believe it may be possible to squeeze checksumming performance down to 500-1000 milliseconds, and for write watched mapped views even lower.

Realistic Performance

As we mentioned in Section 6.1 it is not realistic for a program to only perform system calls. To see how Dragoon fared in the face of this reality we modified the dummy program to do an additional loop in between each system call. The loop is not recorded by Dragoon and spreads each system call invocation out evenly throughout the program execution. This is more in line with what a real program would look like.

Surprisingly we found that by having the program execute for a longer amount of time by simply giving it more code to execute, and not more non-deterministic events for Dragoon to record, that we were able to achieve a recording slowdown nearing 1x.

We can observe from Table 6.1 how the record slowdown is correlated to the amount of space in between each system call. These findings show that given a moderately tuned event log buffer size, programs will be recorded

Loop iterations per system call	Recording slowdown
100	2,8843x
1.000	2,0470x
10.000	1,0432x
100.000	1,0178x
1.000.000	1,0096x

Table 6.1: Recording slowdown factor for varying system call spread intervals.

faster the more time they spend in between system calls. We therefore find it reasonable to expect that on a real program Dragoon should be able to record with a slowdown in the range of 1-4x on average. By tuning the event log buffer size we expect this number to possibly reach as low as 1-2x recording slowdown on average.

It is too soon to make a more educated guess as support for multi-threading is likely to incur some contention, but the current results look hopeful.

6.4.4 Conclusion

We have seen that by manually handling system calls we gain a significant performance boost compared to handling system calls automatically, which was to be expected. It is therefore in Dragoon's best interest to determine which system calls are the most important, e.g., by performing a system call frequency analysis, and manually handle the most frequent system calls. We expect that the remaining system calls will be rarely seen and should account for a small amount, e.g., 0-15%, of the total amount during the program execution. We can therefore expect Dragoon to have a relatively high performance in general, possibly in the range of 1-5x recording slowdown.

Additionally, due to debugging concerns Dragoon has currently disabled function inlining and we therefore expect to see a small performance boost by enabling this at a later time.

We note that by abandoning the current checksumming approach in favor of write watching mapped views behind the scenes, the automatic system call handling performance should see a major performance lift. We also note that even though Dragoon may not get a full score with regards to performance it is one of the only deterministic replay tools in existence that is able to automatically handle system calls, which means it should be able to record

any program if performance is not a concern.

Last to mention is that Dragoon does not rely on operating system support (save for write watching if it is being used) and we consider the observed results to be satisfying overall given this fact. If write watching was to be avoided in order for Dragoon to be a pure user mode implementation, we would see a major reduction in performance. As mentioned earlier in this thesis, however, comparing deterministic replay tools is apples and oranges; if we simply disabled automatic system call handling we would be right back down to a high recording performance again. We therefore conclude that given the circumstances Dragoon's overall performance is acceptable.

Chapter 7

Conclusion

In this chapter we summarize the work presented in this thesis and answer the questions which formed the problem statement in Section 1.2. We then list our main contributions and discuss the potential for future work.

7.1 Summary

In this thesis we have delved into the research area of deterministic replay and found that single-processor user mode deterministic replay tools are not as mature as is presented by modern research. The main trait of current tools is that virtually all of them are implemented for Linux and in addition they rely on help from the operating system to do difficult tasks. Seeing as Windows is less flexible than Linux and lacking in public documentation, we decided to determine whether it was possible to implement a deterministic replay tool on Windows at all.

We began by researching the current state of the art related to deterministic replay and found that multi-processor schemes is where the current focus lies. Single-processor schemes are considered mature and a solved problem, therefore receiving less attention. The general overview of the research field is presented in Chapter 3.

Although hard to come by, we were able to find deterministic replay tools for the Windows platform, thereby answering whether a Windows implementation was possible. Like other deterministic replay tools, the Windows implementations relied on operating system support, something which made them inflexible for out-of-the-box use by the average developer. We therefore

asked ourselves whether it was possible to implement a deterministic replay tool on Windows with no operating system support, in user mode only, something which even previous Linux implementations had neglected. We found that restricting the tool to user mode was a challenging endeavor and have dedicated Chapter 4 to cover our findings and previous research on the topic.

As expected we found that Windows is a restrictive platform to develop determinisitic replay tools on due to lack of documentation and a more restrictive API. Additionally we discovered that previous research has been including operating system support in the definition of user mode, something which challenged our own view of the term. What this means in practice is that no research has yet been able to come up with a design that is a *pure* user mode design, and we therefore made this a requirement in our research. We also noted that other deterministic replay tools have mostly been designed to only support a certain amount of system calls, nor have they been able to achieve very high recording performance in user mode, and we therefore included both of these as requirements in order to see how far we could push our implementation on the Windows platform.

We then started designing the foundation for our deterministic replay tool. A recording prototype and a replay prototype were introduced to experiment with how our ideas worked in practice, and from this we gained an intuition to how we should design the rest of the system. Chapter 5 goes into detail on the design of a complete deterministic replay tool. As a part of this process we were also required to familiarize ourselves with the Windows operating system and certain topics shared by multiple other research fields such as program analysis, malware analysis and reverse engineering. We present an introduction to these concepts and techniques in Chapter 2.

Of particular interest is our proposed solutions to how certain challenging non-deterministic events can be recorded in user mode. We record thread scheduling by forcing all threads to remain in a predefined location (called a *scheduling garage*) while only a single thread gets to run. When the thread gets switched out the new thread moves the last thread to the garage. It then dispatches itself to the code it was running before it was switched out the previous time. The switch itself is recorded from the garage as it is the central point where all threads are forced to pass through before they are allowed to resume their earlier execution.

Non-deterministic instructions are recorded by overwriting their instruction bytes and replacing them with software interrupts. The interrupts are caught in the exception handler, the corresponding non-deterministic instruc-

tion emulated and recorded, then the execution continues as if nothing happened.

We also describe an elaborate method of automatically recording all system calls without knowing their semantics, thus not requiring to have access to their documentation or knowing which side-effects they cause. We do this by comparing the address space before and after every system call, and record any changes.

To enable the replay of the recorded logs we are required to somehow recreate and simulate the address space of the original program, and we describe three ways to do this, two of which use dynamic binary instrumentation in order to outperform the classical software emulation approach. The two methods recreate the address space differently: the first recreates it within its own isolated process whereas the other shares the address space with the replay component while still managing to keep them separate. The former approach suffers from a serious caveat which the other approach solves but in return gives up some performance.

Implementing a deterministic replay tool is a niche area which forbids many of the standard programming practices we normally take for granted, therefore due to the high implementation cost we were unable to complete either prototype. The information gathered during this process however proved invaluable and allowed us to make key decisions with the rest of the design in addition to making educated guesses about the future of the project. We present and discuss our findings in Chapter 6.

7.2 Main Contributions

In this thesis we provide multiple contributions to the field of deterministic replay.

We have provided a summary of the challenges faced when implementing deterministic replay tools in user mode, a somewhat neglected topic, thereby saving other researchers the time of having to find this information for themselves.

We have also implemented many key aspects of the deterministic replay tool design in code, thus showing in detail which steps must be taken in order to implement certain abstract techniques. A concept mentioned in some previous research is that of recreating the address space of the recorded program from the logs, but it has been neglected to describe how exactly this

is done. We show how to achieve this without operating system support and thereby save other researchers a great deal of time as implementing this concept proved to be particularly challenging. In addition to this many of Windows's user mode data structures are poorly documented and our code may therefore serve as documentation to said structures.

We present three novel ideas: recording thread scheduling and non-deterministic instructions without operating system support, and a way to recreate a recorded address space for replay that should outperform the classic approach of software emulation. This technique does not rely on operating system support either.

The thesis answers some key questions:

1. *Is it possible to implement a deterministic replay tool in user mode without operating system support for the Windows platform?*
   Yes. Compared to user mode deterministic replay tools which rely on operating system support this has proved to be challenging, and as a result we had to introduce two novel techniques, but it can definitely be done. The solution presented in this thesis has been generalized as much as possible in order for it to be usable on other platforms as well.

2. *Is it possible to efficiently record arbitrary system calls and their side-effects in an automatic manner without manual handling so that documentation and knowledge of user space internals is not necessary, with no operating system support?*
   The answer to this is both yes and no. It is possible to automatically record undocumented system calls and their side-effects, but the resulting performance may not always be satisfying. This depends on the capabilities of the platform and we have been unable to come up with a generic solution that does not include some risk. However, our solution for Windows should be quite fast.

3. *Is it possible to achieve a recording performance of ≈2x slowdown, which also scales to large programs (1 GB+ RAM), with no operating system support?*
   Yes. Even though our design proposal remains theoretical the observed results of the implemented prototypes show that it is possible to achieve recording performance as low as 1-5x slowdown. Our findings suggest that this should also apply to real programs. We were unable to test the design on programs with high memory usage, but we have successfully

been able to implement infrastructure which works fully in constant time ($O(1)$) and therefore have high hopes for the scalability of our solution.

7.3 Future Work

In this thesis we have attempted to cover every possible angle of how one would implement a deterministic replay tool in user mode. We recognize that if user mode deterministic replay includes operating system support in its definition, then by introducing the constraint of no operating system support we get an entirely new research sub-field that can be explored further. Major questions that still remain unanswered in this area are:

- *How can we record shared memory without using instrumentation?*
- *How can we record non-deterministic instructions in an effective and generic manner?*

The design proposal introduced in this thesis for a Windows implementation is still lacking in many areas. For instance:

- *Can it be extended to record child processes?*
- *Can it be extended to support special Windows features such as thread pooling and Address Windowing Extensions?*
- *Can it be extended to record 64-bit programs?*
- *Can it be implemented on Windows 10?*
- *Can it be extended to support parallel programs?*
- *Can any of the currently used recording or replaying techniques be improved?*
- *Can it be extended to other architectures such as ARM? (the current solution only offers partial support for other architectures)*

We must also consider privacy concerns when implementing a deterministic replay tool. If every executed instruction and memory access made during the program's lifetime is recorded, then what would happen if the recorded

logs fell into the wrong hands? Passwords, credit card information and other personal data would be ripe for the taking, because it would not be a question of whether or not the data had been safely deleted from the program after use; an adversary could easily just go backwards in the recorded execution history and extract the information. These scenarios are particularly relevant for customers submitting recorded program runs for debugging. How can we protect the privacy of the customer while still replaying faithfully? How do we even know the log has not been tampered with? There are many questions to be answered here.

Appendices

Appendix A

Source Code

The source code for the deterministic replay tool designed and implemented in this thesis is located at https://github.com/uioatle/dragoon.

Bibliography

[1] Bowen Alpern et al. "A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications". In: *Proceedings of the 15th International Parallel &Amp; Distributed Processing Symposium*. IPDPS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 23–. ISBN: 0-7695-0990-8. URL: http://dl.acm.org/citation.cfm?id=645609.662644.

[2] Pedro Alves. *GDB, so where are we now? Status of GDB's ongoing target and run control projects*. Feb. 2014. URL: https://archive.fosdem.org/2014/schedule/event/gdb_target_run_valgrind/attachments/slides/393/export/events/attachments/gdb_target_run_valgrind/slides/393/pedro_alves_gdb_slides.pdf (visited on 04/30/2017).

[3] Thomas Anderegg. "Evaluating and optimizing the performance of a user-level record/replay framework". In: (2013).

[4] Piotr Bania. "Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs". In: *CoRR* abs/0905.4581 (2009). URL: http://arxiv.org/abs/0905.4581.

[5] Earl T. Barr and Mark Marron. "Tardis: Affordable Time-travel Debugging in Managed Runtimes". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 67–82. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660209. URL: http://doi.acm.org/10.1145/2660193.2660209.

[6] Arkaprava Basu, Jayaram Bobba, and Mark D. Hill. "Karma: Scalable Deterministic Record-replay". In: *Proceedings of the International Conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: ACM, 2011, pp. 359–368. ISBN: 978-1-4503-0102-2. DOI: 10.1145/

1995896.1995950. URL: http://doi.acm.org/10.1145/1995896.1995950.

[7] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. "A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions". In: *ACM Comput. Surv.* 48.1 (Aug. 2015), 10:1–10:33. ISSN: 0360-0300. DOI: 10.1145/2775111. URL: http://doi.acm.org/10.1145/2775111.

[8] Sanjay Bhansali et al. "Framework for Instruction-level Tracing and Analysis of Program Executions". In: *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 154–163. ISBN: 1-59593-332-8. DOI: 10.1145/1134760.1220164. URL: http://doi.acm.org/10.1145/1134760.1220164.

[9] Bob Boothe. "Efficient Algorithms for Bidirectional Debugging". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 299–310. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349339. URL: http://doi.acm.org/10.1145/349299.349339.

[10] Anton Burtsev. "Deterministic systems analysis". PhD thesis. The University of Utah, 2013.

[11] Raymond Chen. *Another reason not to do anything scary in your DllMain: Inadvertent deadlock*. TODO. URL: https://blogs.msdn.microsoft.com/oldnewthing/20040128-00/?p=40853 (visited on 07/18/2017).

[12] Raymond Chen. *How does the system send non-queued messages?* 1999. URL: https://groups.google.com/d/msg/comp.os.ms-windows.programmer.win32/Hi6yc4-J9ZI/l7XRUFIjgCUJ (visited on 07/26/2017).

[13] Raymond Chen. *Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?* 2011. URL: https://blogs.msdn.microsoft.com/oldnewthing/20110921-00/?p=9583 (visited on 07/22/2017).

[14] Yunji Chen et al. "Deterministic Replay: A Survey". In: *ACM Comput. Surv.* 48.2 (Sept. 2015), 17:1–17:47. ISSN: 0360-0300. DOI: 10.1145/2790077. URL: http://doi.acm.org/10.1145/2790077.

[15] Eugene Ching. *User-mode callbacks in Windows*. 2013. URL: http://eugeii.com/posts/user-mode-callbacks-in-windows/ (visited on 07/20/2017).

[16] D. E. Comer et al. "Computing As a Discipline". In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782. DOI: 10.1145/63238.63239. URL: http://doi.acm.org/10.1145/63238.63239.

[17] Frank Cornelis, Michiel Ronsse, and Koen De Bosschere. "Tornado: A novel input replay tool". In: *In Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03*. CSREA Press, 2003, pp. 1598–1604.

[18] Ryan Cox. *Reading Intel Uncore Performance Counters from User Space*. 2010. URL: http://tech.ryancox.net/2010/11/reading-intel-uncore-performance.html.

[19] Ph.D. David B. Probert. *Windows Kernel Internals: Process Architecture*. URL: http://i-web.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/13-Processes/Processes.pdf (visited on 07/17/2017).

[20] *Debugging with GDB: Process Record and Replay*. URL: https://sourceware.org/gdb/onlinedocs/gdb/Process-Record-and-Replay.html (visited on 04/30/2017).

[21] *Debugging with GDB: Reverse Execution*. URL: https://sourceware.org/gdb/onlinedocs/gdb/Reverse-Execution.html#Reverse-Execution (visited on 04/23/2017).

[22] David Devecsery et al. "Eidetic Systems." In: *OSDI*. Vol. 14. 2014, pp. 525–540.

[23] GDB Developers. *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/ (visited on 04/30/2017).

[24] Brendan F Dolan-Gavitt et al. "Repeatable reverse engineering for the greater good with panda". In: *Technical Report: CUCS-023–14* (2014).

[25] George W. Dunlap et al. “ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay”. In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2002), pp. 211–224. ISSN: 0163-5980. DOI: 10.1145/844128.844148. URL: http://doi.acm.org/10.1145/844128.844148.

[26] J. Engblom. “A review of reverse debugging”. In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. Sept. 2012, pp. 1–6.

[27] Stuart I. Feldman and Channing B. Brown. “IGOR: A System for Program Debugging via Reversible Execution”. In: *SIGPLAN Not.* 24.1 (Nov. 1988), pp. 112–123. ISSN: 0362-1340. DOI: 10.1145/69215.69226. URL: http://doi.acm.org/10.1145/69215.69226.

[28] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. 2004-2017. URL: http://agner.org/optimize/calling_conventions.pdf (visited on 07/19/2017).

[29] Thomas Garnier. *Analyzing local privilege escalations in win32k*. 2008. URL: http://www.uninformed.org/?v=10&a=2 (visited on 07/20/2017).

[30] Dennis Michael Geels et al. “Replay debugging for distributed applications”. In: (2006).

[31] Michelle L. Goodstein et al. “Butterfly Analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring”. In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 257–270. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736050. URL: http://doi.acm.org/10.1145/1736020.1736050.

[32] Julian B Grizzard and Ryan W Gardner. “Analysis of Virtual Machine Record and Replay for Trustworthy Computing”. In: *Johns Hopkins APL technical digest* 32.2 (2013), p. 528.

[33] Zhenyu Guo et al. “R2: An application-level kernel for record and replay”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association. 2008, pp. 193–208.

[34] Jaeseung Ha. *WDD - Deterministic debugging for Windows*. 2017. URL: https://github.com/ipkn/wdd (visited on 07/25/2017).

[35] Steve Herrod. *The Amazing VM Record/Replay Feature in VMware Workstation 6*. Apr. 2007. URL: https://cto.vmware.com/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6/ (visited on 04/23/2017).

[36] Nima Honarmand et al. "Cyrus: Unintrusive Application-level Record-replay for Replay Parallelism". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 193–206. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451138. URL: http://doi.acm.org/10.1145/2451116.2451138.

[37] George Hotz. *USENIX Enigma 2016 - Timeless Debugging*. At time 12:49. Feb. 2016. URL: https://www.youtube.com/watch?v=eGl6kpSajag (visited on 04/30/2017).

[38] howzatt.demon.co.uk. *NtTrace - Native API tracing for Windows*. URL: http://www.howzatt.demon.co.uk/NtTrace/ (visited on 04/23/2017).

[39] Valgrind Developers (http://valgrind.org/info/developers.html). *Nulgrind: the minimal Valgrind tool*. 2000-2017. URL: http://valgrind.org/docs/manual/nl-manual.html (visited on 07/23/2017).

[40] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3A: System Programming Guide, Part 1*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf (visited on 08/01/2017).

[41] Intel. *Pin 2.13 User Guide*. 2013. URL: https://software.intel.com/sites/landingpage/pintool/docs/61206/Pin/html/index.html (visited on 07/23/2017).

[42] IntellectualHeaven. *StraceNT - A System Call Tracer for Windows*. 2011. URL: http://intellectualheaven.com/default.asp?BH=StraceNT (visited on 04/23/2017).

[43] Ekaterina Itskova. "Echo: A deterministic record/replay framework for debugging multithreaded applications". In: *Master's thesis*. 2006.

[44] Ken Johnson. *A catalog of NTDLL kernel mode to user mode callbacks, part 2: KiUserExceptionDispatcher*. 2007. URL: http://www.nynaeve.net/?p=201 (visited on 07/20/2017).

[45] Ken Johnson. *A catalog of NTDLL kernel mode to user mode callbacks, part 3: KiUserApcDispatcher*. 2007. URL: http://www.nynaeve.net/?p=202 (visited on 07/20/2017).

[46] Ken Johnson. *A catalog of NTDLL kernel mode to user mode callbacks, part 4: KiRaiseUserExceptionDispatcher*. 2007. URL: http://www.nynaeve.net/?p=203 (visited on 07/21/2017).

[47] Ken Johnson. *A catalog of NTDLL kernel mode to user mode callbacks, part 5: KiUserCallbackDispatcher*. 2007. URL: http://www.nynaeve.net/?p=204 (visited on 07/20/2017).

[48] Ken Johnson. *A catalog of NTDLL kernel mode to user mode callbacks, part 6: LdrInitializeThunk*. URL: http://www.nynaeve.net/?p=205 (visited on 07/17/2017).

[49] Ken Johnson. *What is the "lpReserved" parameter to DllMain, really? (Or a crash course in the internals of user mode process initialization)*. 2007. URL: http://www.nynaeve.net/?p=127 (visited on 07/20/2017).

[50] Ken Johnson. *Why are certain DLLs required to be at the same base address system-wide?* URL: http://www.nynaeve.net/?p=198 (visited on 07/19/2017).

[51] Mateusz Jurczyk. *Kernel exploitation – r0 to r3 transitions via KeUserModeCallback*. 2010. URL: http://j00ru.vexillium.org/?p=614 (visited on 07/20/2017).

[52] Anand Khanse. *Microsoft Time Travel Tracing Diagnostic Tool*. July 2013. URL: http://www.thewindowsclub.com/microsoft-time-travel-tracing-diagnostic (visited on 04/23/2017).

[53] Samuel T. King, George W. Dunlap, and Peter M. Chen. "Debugging Operating Systems with Time-traveling Virtual Machines". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1247360.1247361.

[54] Dr. Christopher Kruegel. *How To Build An Effective Malware Analysis Sandbox*. Mar. 2014. URL: https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/ (visited on 04/23/2017).

[55] Dongyoon Lee et al. "Respec: Efficient Online Multiprocessor Replayvia Speculation and External Determinism". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 77–90. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736031. URL: http://doi.acm.org/10.1145/1736020.1736031.

[56] Bil Lewis. "Debugging Backwards in Time". In: *CoRR* cs.SE/0310016 (2003). URL: http://arxiv.org/abs/cs.SE/0310016.

[57] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis". In: *CoRR* abs/1601.07962 (2016). URL: http://arxiv.org/abs/1601.07962.

[58] Dominic Lucchetti, Steven K. Reinhardt, and Peter M. Chen. "ExtraVirt: Detecting and Recovering from Transient Processor Faults". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 1–8. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1118621. URL: http://doi.acm.org/10.1145/1095810.1118621.

[59] Chi-Keung Luk et al. "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. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[60] MalwareTech. *Inline Hooking for Programmers (Part 2: Writing a Hooking Engine)*. 2015. URL: https://www.malwaretech.com/2015/01/inline-hooking-for-programmers-part-2.html (visited on 07/22/2017).

[61] Alex Ionescu Mark E. Russinovich David A. Solomon. *Windows Internals, Part 1 (6th Edition)*. Microsoft Press, 2012.

[62] Geoff McDonald and Zelimir Bozic. *Function Hacker 2.3*. 2014. URL: http://function-hacker.software.informer.com/2.3/ (visited on 04/23/2017).

[63] J. M. Mellor-Crummey and T. J. LeBlanc. "A Software Instruction Counter". In: *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS III. Boston, Massachusetts, USA: ACM, 1989, pp. 78–86. ISBN: 0-89791-300-0. DOI: 10.1145/70082.68189. URL: http://doi.acm.org/10.1145/70082.68189.

[64] P. Montesinos, L. Ceze, and J. Torrellas. "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Ef?ciently". In: *2008 International Symposium on Computer Architecture*. June 2008, pp. 289–300. DOI: 10.1109/ISCA.2008.36.

[65] Pablo Montesinos et al. "Capo: A Software-hardware Interface for Practical Deterministic Multiprocessor Replay". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 73–84. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508254. URL: http://doi.acm.org/10.1145/1508244.1508254.

[66] MSDN. *Acquiring high-resolution time stamps*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx#general_faq_about_qpc_and_tsc (visited on 04/30/2017).

[67] MSDN. *An Introduction to Kernel Patch Protection*. 2006. URL: https://blogs.msdn.microsoft.com/windowsvistasecurity/2006/08/12/an-introduction-to-kernel-patch-protection/ (visited on 08/01/2017).

[68] MSDN. *Driver Signing*. URL: https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/driver-signing (visited on 04/30/2017).

[69] MSDN. *Driver Signing Policy*. URL: https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later- (visited on 04/30/2017).

[70] MSDN. *Dynamic-Link Library Best Practices*. TODO. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dn633971(v=vs.85).aspx (visited on 07/18/2017).

[71] MSDN. *HCP Reference*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd796399(v=vs.85).aspx (visited on 04/30/2017).

[72] MSDN. *KeSaveExtendedProcessorState routine*. 2009-2017. URL: https://msdn.microsoft.com/library/windows/hardware/ff553238 (visited on 07/19/2017).

[73] MSDN. *Legacy Floating-Point Support*. 2005-2008. URL: https://msdn.microsoft.com/en-us/library/a32tsf7t(VS.80).aspx (visited on 07/19/2017).

[74] MSDN. *Performance and Memory Consumption Under WOW64*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa384219(v=vs.85).aspx (visited on 07/18/2017).

[75] MSDN. *The TESTSIGNING Boot Configuration Option*. URL: https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/the-testsigning-boot-configuration-option (visited on 04/30/2017).

[76] MSDN. *User-Mode Scheduling*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd627187(v=vs.85).aspx (visited on 04/30/2017).

[77] MSDN. *Using extended processor features in Windows drivers*. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/floating-point-support-for-64-bit-drivers (visited on 07/19/2017).

[78] MSDN. *Using Floating Point in a WDM Driver*. 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-floating-point-or-mmx-in-a-wdm-driver (visited on 07/19/2017).

[79] MSDN. *WOW64 Implementation Details*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa384274(v=vs.85).aspx (visited on 07/18/2017).

[80] Satish Narayanasamy. *Deterministic replay using processor support and its applications*. University of California, San Diego, 2007.

[81] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[82] Robert O'Callahan et al. "Lightweight User-Space Record And Replay". In: *CoRR* abs/1610.02144 (2016). URL: http://arxiv.org/abs/1610.02144.

[83] Hans Passant. *Why must SetWindowsHookEx be used with a windows message queue*. 2011. URL: https://stackoverflow.com/a/7460728 (visited on 07/26/2017).

[84] Harish Patil et al. "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 2–11. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772958. URL: http://doi.acm.org/10.1145/1772954.1772958.

[85] Matt Pietrek. *A Crash Course on the Depths of Win32™ Structured Exception Handling*. URL: https://www.microsoft.com/msj/0197/exception/exception.aspx (visited on 07/17/2017).

[86] *Qira*. URL: http://qira.me (visited on 04/30/2017).

[87] James R. *Processor Tracing*. Sept. 2013. URL: https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing (visited on 04/23/2017).

[88] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. "Detecting System Emulators". In: *Proceedings of the 10th International Conference on Information Security*. ISC'07. Valparaíso, Chile: Springer-Verlag, 2007, pp. 1–18. ISBN: 3-540-75495-4, 978-3-540-75495-4. URL: http://dl.acm.org/citation.cfm?id=2396231.2396233.

[89] "RbMm". *What Does Windows Do Before Main() is Called?* 2016. URL: https://stackoverflow.com/a/41396596/7070908 (visited on 07/20/2017).

[90] "RbMm". *WinApi: Can the message loop be interrupted by an Async Procedure Call?* 2016. URL: https://stackoverflow.com/a/41240391.

[91] mozilla research. *How rr works*. URL: http://rr-project.org/rr.html (visited on 04/30/2017).

[92] Daniel Reynaud. *PIN versus Self-Checking and Self-Modifying Code*. 2009. URL: https://indefinitestudies.org/2009/01/12/pin-versus-self-checking-and-self-modifying-code/ (visited on 07/27/2017).

[93] RS. *Better Software Development with Replay Debugging: Goodbye, Replay Debugging...* Sept. 2011. URL: http://www.replaydebugging.com/2011/09/goodbye-replay-debugging.html (visited on 04/23/2017).

[94] Mark Russinovich. *Pushing the Limits of Windows: Processes and Threads*. URL: https://blogs.technet.microsoft.com/markrussinovich/2009/07/05/pushing-the-limits-of-windows-processes-and-threads/ (visited on 07/18/2017).

[95] Mark Russinovich and Bryce Cogswell. "Operating system support for replay of concurrent non-deterministic shared memory applications". In: *IEEE Computer Society Bulletin of the Technical Commitee on Operating Systems and Application Environments (TCOS)* 7.4 (1995).

[96] Mark Russinovich and Bryce Cogswell. "Replay for Concurrent Non-deterministic Shared-memory Applications". In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 258–266. ISBN: 0-89791-795-2. DOI: 10.1145/231379.231432. URL: http://doi.acm.org/10.1145/231379.231432.

[97] Hex Rays SA. *The PIN Tracer module*. URL: https://www.hex-rays.com/products/ida/support/tutorials/pin/pin_tutorial.pdf (visited on 04/30/2017).

[98] Yasushi Saito. "Jockey: A User-space Library for Record-replay Debugging". In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. AADEBUG'05. Monterey, California, USA: ACM, 2005, pp. 69–76. ISBN: 1-59593-050-7. DOI: 10.1145/1085130.1085139. URL: http://doi.acm.org/10.1145/1085130.1085139.

[99] Open Source. *ReactOS - r75378 - thread.c File Reference*. 2017. URL: https://doxygen.reactos.org/d0/d85/dll_2win32_2kernel32_2client_2thread_8c.html#a22efd4229377caab690bfb682bd9357d (visited on 07/20/2017).

[100] Sudarshan M Srinivasan et al. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging." In: *USENIX Annual Technical Conference, General Track*. Boston, MA, USA. 2004, pp. 29–44.

[101] John Steven et al. "jRapture: A Capture/Replay Tool for Observation-based Testing". In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '00. Portland, Oregon, USA: ACM, 2000, pp. 158–167. ISBN: 1-58113-266-2. DOI: 10.1145/347324.348993. URL: http://doi.acm.org/10.1145/347324.348993.

[102] Undo. *Live Recorder for Automated Test - Reversible Debugging Tools for C/C++ on Linux and Android*. URL: http://undo.io/products/live-recorder-automated-test/ (visited on 04/30/2017).

[103] Undo. *Undo Frequently Asked Questions*. URL: http://undo.io/products/frequently-asked-questions/ (visited on 04/30/2017).

[104] Kaushik Veeraraghavan et al. "DoublePlay: Parallelizing Sequential Logging and Replay". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: ACM, 2011, pp. 15–26. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950370. URL: http://doi.acm.org/10.1145/1950365.1950370.

[105] Ben Voigt. *does not go inside the Windows GetMessage loop on console application*. 2015. URL: https://stackoverflow.com/a/28248281 (visited on 07/26/2017).

[106] wap2k. *Kernel32!BaseThreadInitThunk - Any Infos?* URL: http://www.rohitab.com/discuss/topic/41193-kernel32basethreadinitthunk-any-infos/ (visited on 07/17/2017).

[107] wap2k. *The LdrShutdownProcess and LdrShutdownThread function*. URL: http://www.rohitab.com/discuss/topic/42326-the-ldrshutdownprocess-and-ldrshutdownthread-function/ (visited on 07/17/2017).

[108] Wikipedia. *Address space layout randomization*. TODO. URL: https://en.wikipedia.org/wiki/Address_space_layout_randomization#Microsoft_Windows (visited on 07/18/2017).

[109] Wikipedia. *Client/Server Runtime Subsystem*. 2017. URL: https://en.wikipedia.org/wiki/Client/Server_Runtime_Subsystem (visited on 07/26/2017).

[110] Bi Wu. "Virtualization with Limited Hardware Support". PhD thesis. 2013. URL: http://dukespace.lib.duke.edu/dspace/handle/10161/8255.

[111] Min Xu et al. "Retrace: Collecting execution trace with virtual machine deterministic replay". In: *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*. 2007.

[112] Oleh Yuschuk. *OllyDbg*. Nov. 2000. URL: http://www.ollydbg.de/ (visited on 04/30/2017).