Master's Thesis

Dynamic Tracing of Windows NT Kernel Mode Components

Johannes Passing

Supervisors
Prof. Andreas Polze, Hasso Plattner Institute, Potsdam, Germany
Dr. Martin von Löwis, Hasso Plattner Institute, Potsdam, Germany
Dipl. Inf. Alexander Schmidt, Hasso Plattner Institute, Potsdam, Germany

October 2008

Abstract

Dynamic tracing can be utilized for a variety of purposes, debugging, performance evaluation, and program analysis being amongst them. Although the implementations of respective tracing systems tend to differ sharply, a limited number of tracing *techniques* can be identified which all of these solutions base on. Based on this insight, part I of this thesis discusses these tracing techniques in detail and proposes an appropriate classification scheme. This scheme promises to allow both current and future tracing solutions to be classified based on their usage of these tracing techniques.

In its second part, this thesis discusses NTrace, a dynamic function boundary tracing solution for Windows NT kernel mode components that has been developed as part of this effort. NTrace not only demonstrates how synergies with Microsoft's Hotpatching technology can be utilized in order to achieve safety regarding runtime code modification. It also stands out due to deep integration with the exception handling infrastructure of Windows, *Structured Exception Handling*. With the ability to trace exception unwinds, NTrace is able to yield more precise results than a sheer function entry/exit tracing approach would allow.

By not restricting the usage to customized kernel versions but providing support for retail editions of IA-32 Windows NT, NTrace also promises general applicability. Finally, the performance of NTrace, and the overhead imposed by tracing activity, is discussed in part III, which concludes the thesis.

Zusammenfassung

Dynamische Ablaufverfolgung kann für eine Vielzahl von Zwecken eingesetzt werden, wie etwa zu Performance-Messungen, Programmanalyse oder zur Fehleranalyse auf Produktivsystemen. Obschon die Implementierungen entsprechender Ablaufverfolgungs-Systeme sich bisweilen stark zu unterscheiden neigen, so kann doch eine kleine Menge von *Techniken* identifiziert werden, auf denen diese basieren. Ausgehend von dieser Erkenntnis befasst sich Teil I dieser Arbeit mit den Details dieser Techniken und schlägt ein Klassifikationsschema für diese vor. Das Schema verspricht, sowohl derzeitige als auch künftige Ablaufverfolgungs-Systeme anhand der von ihnen verwendeten Techniken einordnen zu können.

Teil II der Arbeit befasst sich mit NTrace, einem Werkzeug zur dynamischen Ablaufverfolgung von Funktionsein- und austritten in Windows NT Kern-Komponenten. Es handelt sich hierbei um eine Neuentwicklung, die im Rahmen dieser Arbeit entstanden ist. NTrace zeichnet sich dabei insbesondere durch zweierlei Aspekte aus: Zum einen wird aufgezeigt, wie Synergien mit der Microsoft Hotpatching-Technologie genutzt werden können, um sichere Anwendung von selbstmodifizierendem Code zu gewährleisten. Zum anderen umfasst NTrace eine tiefgreifende Integration mit der Ausnahmebehandlungs-Infrastruktur des NT Kerns, *Structured Exception Handling*. Diese Integration ermöglicht NTrace die Ablaufverfolgung von Ausnahmeabwicklungen, um so präzisere Ergebnisse zu erlangen.

NTrace erfordert keinerlei Anpassungen am NT Kern und kann somit auf handelsüblichen Editionen des Windows NT IA-32 Kerns genutzt werden, was wiederum vielfältige Anwendungsmöglichkeiten verspricht. Eine Betrachtung der Leistung von NTrace und dem mit der Ablaufverfolgung verbundenem Overhead ist Teil von Teil III, welcher diese Arbeit abschließt.

Acknowledgments

For their valuable feedback, perceptive criticisms and fruitful discussions, I would like to thank the supervisors of this thesis, Martin von Löwis and Alexander Schmidt.

Not only their support, but also having created the opportunity for me to work with the Windows NT kernel and address the technically challenging topic of dynamic tracing has been highly appreciated.

Contents

List of Figures

List of Tables

Listings

Part I

Theoretical Groundwork

1 Introduction

With growing complexity, software systems have become increasingly hard to understand and debug. Confronted with large volumes of source code, it is challenging for a developer to reason about the potential behavior of a system at runtime, much less to diagnose erroneous behavior exhibited by a running system. Having appropriate tools at hand is hence indispensable.

This thesis covers dynamic tracing, a building block for creating tools helping to analyze systems by observing their bevavior. Observing a program as it is being executed, and not requiring it to be rebuilt or restarted, dynamic tracing is particularly well-suited for long running systems such as operating system kernels. The focus of this thesis therefore lies on the analysis of the Windows NT kernel and its components.

The topic can be further refined as addressing function boundary tracing, i.e. capturing function entry and exit events. Software tracing can be performed at an even more fine-grained level, yet, function boundary tracing may be expected to yield rather universally applicable results for the aforementioned purposes.

1.1 Potential Fields of Applications

Given this background, potential and appropriate fields of applications for a dynamic tracing system for the Windows NT kernel can now be laid out more clearly.

An important potential field of application is production debugging. In production debugging scenarios, a dynamic tracing solution promises to allow instrumentation and recording of information without interrupting the service of the examined system. Once information has been collected, instrumentation can be revoked and the data can be analyzed *offline*.

Besides the capability of capturing information valuable to pinpoint flaws in a software system, a dynamic function boundary tracing system also promises to help in comprehending a system. Traces, in particular when visualized properly – for instance, as call graphs – can help illuminating the runtime behavior and dynamics of a software system.

Dynamic tracing can also be used for profiling. In contrast to sampling-based profilers such as *kernrate*[1], such a profiler could potentially be more accurate as the risk of *missing* a function call due to the sampling interval chosen ceases to exist.

In a similar manner, code coverage – albeit limited to function level rather than basic block level – could be implemented based on such a dynamic tracing solution.

1.2 Structure of this Thesis

The thesis is structured into three parts. Part 1 covers the theoretical groundwork of the topic. This includes an assessment and classification of existing dynamic tracing approaches, along with a discussion of their strengths and weaknesses. The first part concludes with a detailed discussion of common implementation challenges related to runtime code modification-based tracing approaches.

Part II introduces NTrace, a dynamic tracing solution developed as part of this thesis. The part opens with a general discussion of the tool architecture and continues by laying out the basic approach to function boundary tracing taken. Based on this groundwork, the details of the approach as well as the implementation challenges are discussed.

The thesis concludes with Part III, which provides an analysis of NTrace. This includes a discussion and measurements of the performance characteristics of the implementation.

[1] Part of the Microsoft Windows Server 2003 Resource Kit

1.3 Contributions

The main contributions of this work are as follows:

- Definition of a classification scheme for dynamic tracing techniques.
- Proposal of a dynamic function boundary tracing system for the Windows NT kernel that fully integrates with the Windows Structured Exception Handling infrastructure.
- Proving the feasibility of the proposed system by an implementation and evaluating its runtime performance.

1.4 Definition of Terms

The semantic differences between the terms function, procedure, and routine are insignificant in the context of this work and the terms are used interchangeably. Following [RR03], the terms *module* (or *executable module*) and *image* (or *program image*) are distinguished as follows: A module constitutes a file, usually generated by a linker, containing executable code. When the module is loaded into a process, the in-memory representation of the module is referred to as *image*.

For the remainder of this work, code is considered *unmodified* if it matches the code that would be executed in the absence of a tracing system.

The terms *trampoline*, *springboard* and *bounce*, all commonly encountered in the literature [BH99, HMC94, Bru04, PKFH02, TM99], will be jointly referred to as *trampoline*.

Any machine that is capable of concurrently executing more than one thread is referred to as multiprocessor machine. This includes true multiprocessor machines as well as machines using multicore CPUs or CPUs supporting technologies such as *Simultaneous Multithreading* [EEL+97].

The terms *i386* or *x86* are avoided in favor of the term *IA-32* [Int07a] to denote the Intel 32 bit architecture. The *AMD64* [Dev07] and *Intel64* [Int07a] architectures are jointly referred to as *AMD64*.

All references to Windows refer to the Windows NT family exclusively. Unless stated otherwise, all discussions of Windows NT further refer to Windows Server 2003 SP1/SP2 and the Windows Research Kernel 1.2 only. The term *kernel* as used in this work does not include drivers but only refers to the functionality implemented in the kernel module, usually named ntoskrnl.exe. In contrast, *kernel mode Windows* refers to the entirety of code running in kernel mode.

Unless stated otherwise, all references to *compiler* and *linker* shall denote the Microsoft compiler *cl* and linker *link*, respectively. As Microsoft compilers and linkers are the prevalent tools for kernel mode Windows development, other compilers are not addressed by this work.

Occasionally, use of the syntax `module!Function` is made to fully qualify routine names. Finally, all assembler code listings use the Intel syntax.

2 Criteria

To allow a more effective assessment of existing and potential tracing techniques, a set of basic criteria such techniques are to meet shall be defined.

In their paper *Dynamic Program Instrumentation for Scalable Performance Tools*, Hollingsworth et al. [HMC94] state that monitoring the performance of massively parallel programs requires an instrumentation system that is *detailed*, *frugal*, and *scalable*.

The focus of this work is not massively parallel systems but rather general purpose systems – still, these three criteria remain valid and provide a suitable foundation. However, to emphasize the fact that monitoring or tracing activity should not endanger system stability, *Robustness* shall be introduced as a fourth criterion.

Detailed

In the realm of function boundary tracing, the desired level of detail is defined by function calls. A solution capable of tracing function entries can be expected to already produce valuable results. Yet, it is desirable to have the tracing solution also support function exit tracing, as this additionally allows inferring call relationships among functions.

Whether more fine grained information such as parameter and return values are of importance depends on the individual purpose. If the intent of tracing is to collect performance information or to comprehend the overall system, such additional information may be not of interest. For problem analysis, in contrast, this additional information may be valuable indeed.

To keep the discussion general in this regard, the aspects of more fine grained information are therefore ignored in the remainder of this work.

Frugal

Frugality – and performance in particular – is at the core of a dynamic tracing solution. In fact, the promise of low runtime overhead may have been the reason for a user deciding for a dynamic tracing solution in the first place. More generally, economical usage of resources is important for limiting the impact on resource and timing behavior of the traced system, and thus, to ensure applicability of the tracing system.

Rather than striving at high efficiency, frugality can also be attained by limiting the amount of data collected. This, however, may stand in conflict with the demand to be detailed. As such, this conflict depicts the classical conundrum of having to balance between performance and precision.

Another aspect of frugality is the runtime overhead in case all tracing activity has been disabled. That is, the tracing system is active, yet no tracing information is currently being captured. In this case, it is desirable for the tracing solution to impose no or at least no significant runtime overhead.

Scalable

Scalability describes the ability of the tracing system to remain frugal and detailed when resources are added. The most important resources in this regard are additional functions to be traced, and additional processors.

When tracing information is to be captured for a growing number of functions, the amount of data that is to be handled will grow as well. To remain frugal, the system must be able to efficiently handle these increased amounts of data.

As multiprocessor machines – and multicore processors in particular – become increasingly common, the ability to properly deal with more than one processor becomes crucial as well. Certainly, this requires the tracing system to be written in a multiprocessor-safe manner. However, to remain frugal, this also requires that the tracing system makes effective use of the additional processors. In particular, it should not limit the degree in which the tracee makes use of the additional processors and it should not impose significant performance bottlenecks itself.

Robust

When tracing is used in development scenarios, robustness may not be of utmost concern. In production debugging scenarios, however, it is crucial that tracing does not endanger stability and availability of the traced system.

Runtime code modification, a technique commonly used for implementing dynamic tracing, is particularly sensitive in this regard. As will be covered in more depth in the remainder of this work, significant attention to the details have to be paid in order not to undermine the stability of the system.

As such, robustness clearly is an important factor to consider when assessing dynamic tracing solutions.

3 Classification of Dynamic Tracing Techniques

In contrast to static tracing, dynamic tracing is based on the idea of allowing trace collection of running programs and unmodified binaries. Any necessary steps to enable such collection is therefore performed at runtime.

Given this rather broad definition of dynamic tracing, a variety of approaches for implementing dynamic tracing can be identified by studying existing research papers and implementations. The individual concepts and algorithms encountered among these approaches vary, yet a small number of key ideas shared among groups of approaches can be identified. Based on this insight, this section proposes a classification of dynamic tracing techniques.

Although the focus of the remaining discussion lies on kernel mode instrumentation, user mode instrumentation solutions are discussed as well. Moreover, approaches targeting broader application than just tracing – such as profiling or dynamic optimization – are discussed for sharing many properties with dynamic tracing approaches. Virtualization-based approaches, however, in which the entire operating system is run in a virtual environment are considered out of the scope of this work.

Any tracing solution, regardless of the level of detail supported, relies on the consumption of events, although the nature of the individual events of interest may vary. However, a first rough distinction of techniques can be made based on the source of these events, which is either hardware or software.

Each of the classes will be discussed individually in the following sections. Along with this discussion, a non-exhaustive list of solutions implementing the individual technique is presented.

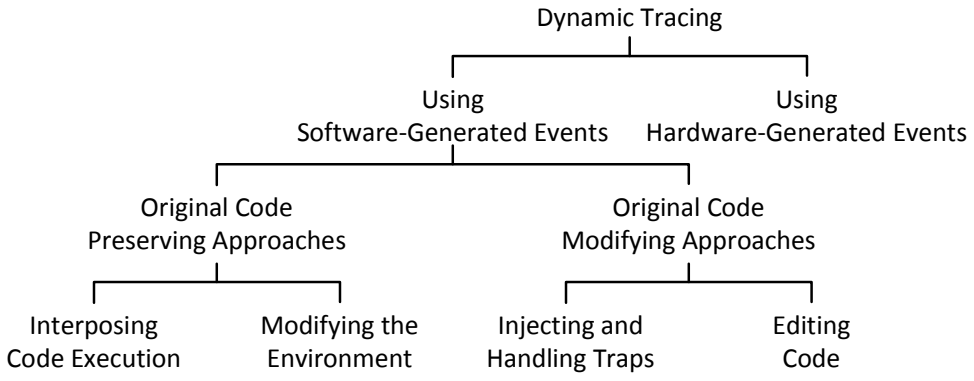

Figure 3.1: Classification of Dynamic Tracing Techniques

It is worth pointing out that the classification refers to *techniques* rather than solutions. Although the majority of tools and solutions leverages a single such tracing technique only, there are solutions that combine two or more of the presented techniques.

3.1 Using Hardware-Generated Events

Definition

A tracing technique that utilizes events generated by the CPU while executing unmodified code.

Discussion

Current processors such as those of the IA-32 family offer features for generating and recording a number of performance and tracing-related events. Although current IA-32 processors do not offer a means to specifically record subroutine calls, they do allow the generation and recording of more fine grained events. A notable feature in this context is *last branch recording* [Int07b], which records origin and target of the last branches taken. Using this and similar processor features, a tracing solution could also deduce more coarse-grained tracing information such as procedure-level traces.

As the collection of tracing information is based on dedicated hardware features, no code has to be modified. In particular, although this technique may require handling of traps, the code does not need to be augmented by trap-generating instructions.

Implementations

The facilities provided by CPUs that may be used for observing execution flow tend to generate very fine-grained information, often on an instruction or branch-level basis. Moreover, features such as last branch recording are rather new additions to the IA-32 instruction set and cannot be expected to be widely supported yet. Still, solutions for procedure-level execution analysis can be identified to rely on such events, although their number seems to be limited.

One of the most common tools that rely on hardware events are sampling profilers. However, the use of sampling to obtain tracing and performance information is arguable due to the inherent risk of impreciseness – a topic that has been discussed extensively in [Tam01]. Yet, these tools rely on frequent timer interrupts, which are clearly events generated by hardware rather than induced by software. In the context of the NT kernel, a notable example of a facility that uses sampling is the built in profiling facility [Neb00], which serves as the basis for the kernrate tool.

Linux kernel version 2.6.25, which was in the state of a release candidate at the time of writing, introduces an enhancement for *ptrace* which makes use of branch trace storage [Int07b] on IA-32 processors [Mol08].

3.2 Using Software-Generated Events

Definition

A tracing technique that utilizes events caused by software. As unmodified code is assumed not to generate the necessary events, adaptions to the code or the way the code is executed is usually required.

Discussion

To allow a more precise discussion of the ideas behind this class of tracing techniques, this class is further broken down into *Original Code Preserving Approaches* and *Original Code Modifying Approaches*. These classes will be discussed separately.

3.3 Original Code Preserving Approaches

Definition

A tracing technique that causes the generation of events necessary for the collection of tracing information without performing in-place modifications on the original code.

Discussion

In order to provoke the generation of such events without modifying affected code, two approaches can be taken. On the one hand, the execution flow of the code can be adapted by changing its environment, e.g. by performing changes on data. On the other hand, different code may be executed, which may or may not be derived from the original code.

The following two sections discuss both approaches.

3.4 Modifying the Environment

Definition

A tracing technique that does not rely on modifying code but rather on altering the environment the code is executed in with the intent of adapting the behavior of the software in a manner that allows tracing information to be gathered.

Discussion

The most prominent example of modifications on the environment that affect execution flow is exchanging the values of memory locations storing addresses used for indirect calls. By altering these locations, a branch into tracing code may be injected which, besides collecting the desired information, delegates to the original target. Figure 3.2 illustrates this idea.

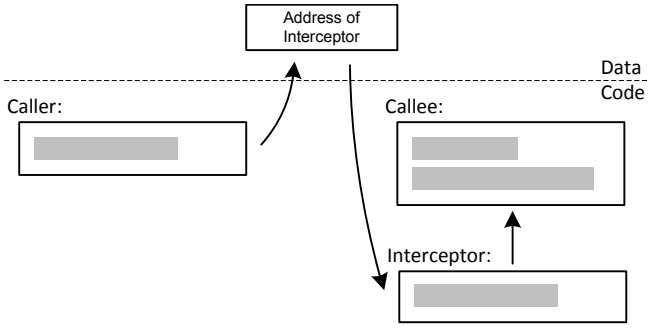

Figure 3.2: Intercepting an indirect call

Whether this approach is applicable or not significantly depends on the individual software to be inspected. Code that does not make use of indirect jumps or whose execution flow cannot be adapted by data modifications in a sufficient manner may not be traceable with such an approach. In contrast, a piece of software may explicitly have been designed for allowing this kind of observation. Such software may, for example, make extensive use of *jump tables*.

The majority of software may be expected to fall in between these two extremes – these systems have not been explicitly designed for allowing tracing but for other reasons make use of programming techniques that can be leveraged for this intent. Examples for such

techniques include *Vtables* as used by C++, COM [Box98] and other object oriented languages and frameworks. Jump tables are very similar to Vtables in that they also store function pointers, although they are usually unrelated to the concepts of object orientation.

A prominent example for a jump table is the *Import Address Table* (IAT) defined by the *Portable Executable File Format* [Cor06b]. The role of the IAT is to serve as the connector between dynamically loaded modules. Whenever a module such as a DLL imports routines from different modules, its IAT will provide one slot for each import. When the respective module is loaded, the loader will resolve these imports and will store the pointers to the imported routines in the corresponding slots of the IAT.

The *Procedure Linkage Table* (PLT) defined by the *Executable and Linking Format* (ELF) [Com95] employs a similar mechanism. Like the IAT, the PLT is used to allow function calls to be made from one executable or shared object to another.

Without restricting this discussion to a specific software package that is to be instrumented, the general applicability of such approaches is therefore hard to quantify, but can at least be expected to be significantly below approaches relying on code modification.

Implementations

The NT kernel makes extensive use of function pointers. While most of these can be expected not to be designed for the purpose of intercepting or tracing operations, some of them are. A prominent example of such *hooks* are those used by Driver Verifier [Cor08c].

Microsoft Driver Verifier is a tool for detecting common flaws in drivers. To detect certain erroneous operations, Driver Verifier has to observe a driver on a rather fine grained level. Among the techniques used by Driver Verifier to attain this observation is to intercept a number of routine calls. For this purpose, the NT kernel deliberately uses function pointers at certain places with the intent of enabling Driver Verifier to hook into calls. An example for such an operation is calling a driver, as implemented by `IofCallDriver`.

As listing 3.1 suggests, the global variable `pIofCallDriver` is NULL during normal operation. To allow hooking, the variable can be set to point to a routine such as `IovCallDriver` (part of Driver Verifier). Besides pre- or postprocessing the call, such a routine will usually delegate the call to `IopfCallDriver`.

```
1  NTSTATUS
2  FASTCALL
3  IofCallDriver(
4      IN PDEVICE_OBJECT DeviceObject,
5      IN OUT PIRP Irp
6      )
7  {
8    if (pIofCallDriver != NULL) {

10     //
11     // This routine will either jump immediately to
12     // IovCallDriver or IoPerfCallDriver.
13     //
14     return pIofCallDriver(
15       DeviceObject, Irp, _ReturnAddress());
16   }

18   return IopfCallDriver(DeviceObject, Irp);
19 }
```

Listing 3.1: Implementation of IofCallDriver, WRK 1.2, BASE\NTOS\IO\IOMGR\iosubs.c, line 2237

As empirical analysis reveals, *IRPTracker* [Res03], a tool that allows tracking of *I/O Request Packets* (IRPs) traveling through the I/O subsystem, also leverages this (undocumented) facility to gather tracing information.

Another common technique used for intercepting routine calls – either to adapt behavior or to implement tracing – is *Import Address Table Hooking* [Rob03]. By replacing a function pointer in the IAT with a pointer to an appropriate tracing routine, inter-module calls can be traced. However, any calls not crossing a module boundary or using function pointers obtained dynamically (such as by using `GetProcAddress`) cannot be easily intercepted with IAT hooks. Notwithstanding these limitations, applied to core OS libraries such as ntdll.dll or kernel32.dll, IAT hooks are a powerful technique for observing the interaction between a user mode program and the operating system. As demonstrated by Leman [Lem00], this approach is not only applicable in user mode but also in kernel mode.

Clowes [Clo01] has shown that a similar technique can be employed to leverage the ELF PLT in order to intercept function calls.

A related, yet more specialized technique is hooking the *System Service Descriptor Table* of the NT kernel, as first published in [RC97]. In a similar manner, calls to interrupt service routines can be intercepted and delegated [HB05, PDB99]. However, these techniques belong to the practices explicitly discouraged by Microsoft [Cor07]. More kernel mode function pointer-based techniques have been discussed in the context of security research in [MJ07].

The *COM Universal Delegator* [Bro99a, Bro99b] defines a method call interception framework for the *Component Object Model* (COM). The basic idea behind this approach is to leverage the fact that all methods of a COM interface are virtual and method invocations are dispatched through a Vtable. In order to intercept all method invocations of a respective interface, the pointers to the methods stored in the Vtable are replaced by pointers to specific *thunk* routines. After preprocessing a call, a thunk routine will remove itself from the stack and delegate the call to the original method implementation. Using return address replacement and a thread local private stack, the framework is also capable of additionally intercepting method returns in order to post-process a call.

A similar approach that additionally allows cross-process tracing of COM method invocations has been published in [Lem99].

As published, both approaches only apply to COM and user mode. However, the basic idea of implementing virtual method dispatching by using Vtables is equally applied in other programming environments. Therefore, this approach could be adapted to be applicable to other scenarios and environments, including kernel mode Windows, as well.

3.5 Interposing Code Execution

Definition

A tracing technique that relies on interposing the code to be traced. Rather than letting the affected regions of original code run natively, code is treated as data and used as information for interpreting or derivation of new code.

Discussion

The basic idea behind these solutions resembles virtual machines in that they rely on techniques such as interpreting code or using *just in time compilation* techniques to derive new code from the original code. In both cases, the software gains the option to intercept certain operations or to augment the code by instrumentation code on the fly.

The latter technique, fetching original code fragments, augmenting it and assembling new code fragments is commonly referred to as *dynamic compilation* [CK94]. The resulting code fragments, called *translations*, are encoded using the same instruction set as the original code. This differs from *dynamic binary translation* [CM], which describes a similar technique, yet involves translating between different instruction sets.

Dynamic compilation offers immense flexibility with regard to the level of detail at which instrumentation can be performed. With regard to function boundary tracing, this flexibility allows the user of such a solution to make a very specific choice of how a routine call should be traced. On the one hand, the event of *calling* a routine could be captured by identifying and instrumenting all code sequences that call the respective routine. On the other hand, the tracing could be performed at the callee's site, i.e. the event of a routine *being called* could be captured by instrumenting the respective routine itself.

Both of these options have their own advantages and disadvantages. On the one hand, as there is usually more than one potential caller of a given routine, tracing on the caller's site offers the ability to further distinguish between calls. For instance, by instrumenting specific callers only, tracing can be scoped to affect only those routine calls performed from specific locations in code. On the other hand, when all calls are to be captured – regardless of their origin – a potentially large number of calls must be properly instrumented. In contrast to that, callee-site instrumentation only requires the routine itself to be instrumented and guarantees all calls to be captured.

In practice, however, caller-site tracing is often limited by the fact that determining all potential callers of a given routine is challenging due to the existence of indirect jumps and calls.

Implementation

Shade [CK94] was among the first tools to implement dynamic compilation. Rather than being launched directly, an application that is to be instrumented is run via Shade. That is, a user launches Shade and requests it to load the respective binary. Although this approach prohibits attaching to a natively running process after the fact, it gives Shade tight control over the execution of the target. Shade uses this control to avoid all execution of native, unmodified code. Instead, Shade makes extensive use of dynamic compilation and only executes the dynamically compiled, instrumented code.

As part of this dynamic compilation, Shade allows calls to *trace functions* to be injected before certain instructions. Using such callbacks, tools such as call profilers and call graph analyzers can be built on top of Shade.

By using appropriate caching mechanisms, Shade aims at compensating the additional overhead of instrumentation and attains reasonable performance. Shade allows instrumentation of user mode SPARC binaries and has been implemented for Solaris.

Dynamo [BDB00] focuses on dynamic optimization – its aim is to transparently improve execution speed of just-in-time generated or even statically optimized code. The basic strategy of Dynamo is to start off as a machine code interpreter. Observing program behavior, Dynamo is capable of identifying *hot*, i.e. frequently executed code regions, which it will attempt to optimize. Acting on the level of *traces*, i.e. sequences of consecutively executed instructions, Dynamo makes use of dynamic compilation facilities similar to Shade to create optimized versions of these traces, so called *fragments*. To account for the frequent use of these code regions, these fragments are cached.

Dynamo has been implemented as a shared library for the PA-8000 architecture and allows being attached to a running process. Although the approach would allow further

instrumentation for purposes such as tracing, Dynamo does not offer such facilities by itself. Still, Dynamo has delivered the groundwork for DynamoRIO, which allows both optimization and instrumentation of applications.

DynamoRIO [Bru04] is a dynamic code manipulation solution for user mode applications on Linux and Windows. DynamoRIO allows runtime manipulation of unmodified binaries for purposes such as tracing. DynamoRIO is similar to Shade in that it also avoids execution of the original code altogether. The code is merely used as a blueprint for deriving, i.e. dynamically compiling, new code. Like Shade, DynamoRIO also has to be loaded during process startup, i.e. before any of the original code has become subject to execution. Attaching to an already running process is thus not possible.

As part of the compilation, code can be specified to be augmented by additional instrumentation code. Moreover, in order to improve execution speed, DynamoRIO is capable of applying certain optimizations to this code. Once compiled, the code fragments are placed into a code cache, from which it can repeatedly be fetched.

Based on this setup, DynamoRIO is able to dynamically instrument an application without the kernel or the application itself being aware of. DynamoRIO provides a rich set of interfaces that allow execution observation of the software on various levels, making it a system whose potential applications reach far beyond tracing on the level of routine calls.

However, to attain this high degree of flexibility, DynamoRIO, like Shade, has to have tight control over the communication between the kernel and the process. Besides system calls, this also includes interfaces such as user mode callbacks invoked by the kernel. While uncommon on Unix systems, such callbacks play an important role on Windows NT. Notwithstanding the complexity involved, DynamoRIO presents a solution that is capable of interposing all such interfaces in order not to lose control over the process.

Valgrind [Net04] is a user mode instrumentation framework for Linux/IA-32. It is designed to serve as the foundation for analysis tools such as profilers and memory checkers. Technically, Valgrind translates the IA-32 machine code from the program to be run to an intermediate representation called *UCode*, at which instrumentation is applied. The instrumented intermediate code is then translated back to IA-32 machine code which is finally executed. This instrumentation is performed lazily on a basic block level.

Similar to Shade and DynamoRIO, Valgrind attempts to avoid the execution of original code. To attain the necessary degree of control over code execution, Valgrind has to be attached during process startup as well. Like DynamoRIO, Valgrind interposes all major communication channels between the operating system kernel and the actual application code, including system calls and signal delivery.

Tools such as *memcheck* and *cachegrind* (part of the Valgrind Tools Suite [Val07]) have shown the effectiveness and flexibility of Valgrind, which these tools are based on. The fact that Valgrind has to be loaded into the affected process from the start is also of minor concern in the context of such tools.

Considering Valgrind, Shade and DynamoRIO for the purpose of dynamic instrumentation and tracing of running systems, however, the requirement of having to be loaded during process startup can turn out to be an issue.

Pin [LCM+05] is an instrumentation solution that shares several ideas with the aforementioned solutions. Pin allows instrumentation of user mode Linux and Windows processes and instruments code in a *just in time* manner. Yet, unlike Shade, DynamoRIO and Valgrind, Pin is also capable of being attached to process after it has started and being detached without the process having to be terminated.

While Pin is limited to user mode, Olszewski et al. [OMCB07] have brought the ideas of just in time instrumentation and late attachment to kernel mode. Sharing basic ideas with Pin, the presented solution allows instrumentation of kernel mode code on the Linux operating system.

Both approaches perform instrumentation by injecting additional code sequences while leaving the original code unmodified: Basic blocks are fetched from their original location, instrumented, compiled, and placed into a code cache as they become subject to execution. Whenever a new block is compiled, the branches from and to this block are updated. Any branches pointing to blocks not yet compiled are linked to a special *dispatcher stub* which, when invoked, will trigger the just in time instrumentation of the respective block.

Like the other solutions discussed in this section, Pin heavily relies on disassembly of the code to be instrumented in order to identify the basic blocks.

However, having execution repeatedly branch between original code, dynamically compiled code blocks, the dispatcher and instrumentation code presents an additional problem: The contents of certain registers may have to be preserved and later restored whenever such a branch is taken. One option is thus to save and restore all CPU registers in these situations, which is guaranteed to be sufficient, yet, induces a non-negligible overhead. A more efficient, yet more challenging approach to implement is thus to perform register liveness analysis – a technique used by several solutions presented in this section, including Pin. By analyzing the usage of registers beforehand, state saving can be limited to *live* registers, i.e. registers known to be referenced at a later stage.

Moreover, for any dynamic compilation technique that allows late attachment to become effective, there has to be at least one point where control is initially transferred to the dispatcher of the instrumentation solution. Once having gained control, the dispatcher can then direct further instrumentation. To achieve this initial takeover of control, these approaches therefore need some jump aid. Pin uses the *ptrace* infrastructure for this purpose while the solution proposed by Olszewski et al. facilitates an additional instrumentation technique, namely using environment modification (see section 3.4): it replaces entries of the system call table.

By writing the dynamically compiled code sequences to a new location and leaving the original code untouched, the challenges and limitations of in place code modifications are largely avoided. An implication of this approach, however, is that the process maintains up to two copies of each block – the dynamically compiled, and the original code block. Disregarding the increased memory requirements, this coexistence of instrumented and uninstrumented code as well as the necessity for providing explicit entry points comes along with both notable advantages and disadvantages.

On the one hand, it may be advantageous that the instrumented piece of code (i.e. a sequence of instrumented basic blocks) is only executed when the execution flow originates from a certain entry point, such as a system call. Tracing all memory allocations made by a specific system call may be an example for such a use case. Not only are memory allocations made by other system calls not traced, the overhead of instrumentation is also only paid for those execution flows that are indeed of interest.

On the other hand, this scoping may well turn out to be a disadvantage. If, for example, all memory allocations – regardless of whether made in the context of a system call or not – are to be traced, such a solution can be inappropriate. More generally, the effectiveness of the approach heavily depends on the area of interest, (i.e. the code of which execution is to be traced), and the question whether it can be fully covered by one or more of such entry points.

Finally, solutions that do not support attachment to a running process do not fully qualify as being *dynamic* in the sense expressed in the beginning of this section. Those solutions that do allow late attachment usually require some kind of bootstrapping. The applicability of such solutions is therefore also dependent on the flexibility of the *jump aid* solution employed. The more entry points this solution allows, the greater the applicability of the solution will be.

3.6 Original Code Modifying Approaches

Definition

In order to generate the events necessary for tracing, in-place modifications on the previously unmodified, original code are performed. The code is augmented so that the necessary events are issued. Such events may include traps or callback routines being invoked.

Discussion

Instrumenting code requires augmenting the code by additional instructions – instructions that, for instance, capture tracing information. The basic problem all code modifying solutions therefore have to face is how these additional instructions can be woven into the original code. This question becomes even more relevant when instrumentation is to be performed on a very fine grained level such as on the level of basic blocks or instructions.

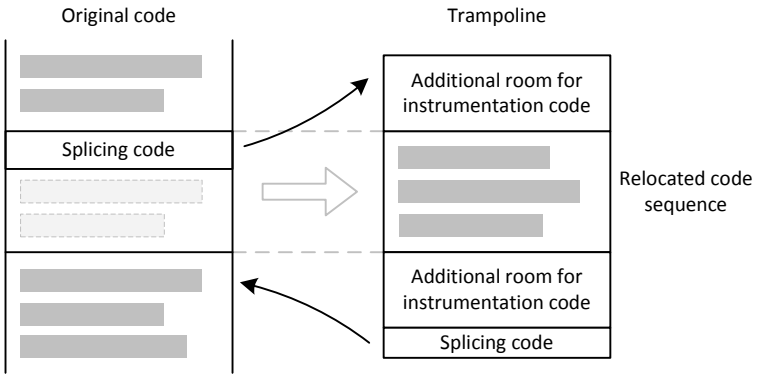

Figure 3.3: Runtime code splicing

One pattern encountered in several solutions is *runtime code splicing* [Thi99]. The idea of runtime code splicing, illustrated in figure 3.3, is as follows: One or more instructions are *cut* out of the original code. These instructions, along with the necessary instrumentation code, are moved to a newly allocated code block, usually referred to as *trampoline*. The crucial point to notice is that the trampoline usually is significantly larger than the piece of relocated code. The original code and this code block are then *spliced* so that the execution flows from the original code to the trampoline and back to the remainder of the original code. Splicing can be applied on any level ranging from single instructions (i.e. only a single instruction is relocated) to entire routines. However, the relocation of code sequences has further ramifications such as the necessity to update all relative addresses used by the relocated instructions.

Especially in the context of runtime code splicing, the differences between original code modifying techniques and code interposing techniques seem to become blurred. The characteristic difference, however, can be found in the fact that the original code remains the scaffold for code splicing solutions: significant parts of the code may have been relocated,

yet, barring exceptions, execution flow is usually routed back to the original code at the end of each such code block.

In contrast, the original code mostly loses its role as serving as the scaffold in the case of solutions implementing code interposition: the default is that execution is not routed back to original code when the end of such a dynamically allocated code block has been reached – rather, execution is routed to the next dynamically compiled block. Execution being routed back to original code is an situation that either never occurs or can at least be considered a comparatively rare occurrence.

When used for tracing of routine calls, both caller-site and callee-site could be implemented using code modification. In practice, however, instrumenting at the callee site is encountered significantly more often.

Regarding existing instrumentation solutions in more detail, it is notable that a significant number of solutions implementing code modification primarily rely on the use of trap-generating instructions. Although many properties are shared among such trap-centric solutions and other, non trap-centric code modification solutions, a separation into two techniques is worth being made.

3.7 Injecting and Handling Traps

Definition

A tracing technique that relies on in-place modification of the original code. In order to be able to interrupt, control, or trace execution, trap-generating instructions are injected into the original code. Using an appropriate trap handler, the events generated by such instructions can be handled accordingly.

Discussion

Injecting trap-generating instructions like `int 3` on IA-32 is the classical approach taken by debuggers to implement breakpoints. As dealing with traps is at the core of hardware and an operating system's capabilities, this approach can be expected to be feasible on a very wide range of operating systems and hardware architectures.

On variable-length instruction set architectures such as the the IA-32, relying on trap-generating instructions has the additional advantage that such instructions can be as short as one byte. With such short instructions, several of the challenges concerning runtime code modification can be circumvented – a topic that will be discussed in more detail in section 4. A drawback of this approach, however, is the non-negligible overhead associated with the handling of traps.

Implementations

As mentioned before, debuggers are the most prominent example of tools using this technique for implementing breakpoints. Current Windows debuggers such as WinDBG also offer tracing facilities, which make them relevant in this discussion. However, as their name already suggests, *breakpoint commands*, which are used for such tracing purposes, also rely on handling of debug traps.

Beyond the realm of debuggers, usage of this technique for tracing purposes can be found in *Dynamic Trace*, a diagnostic facility introduced by OS/2 Warp 4 fixpack 4 [Cor97]. By injecting an `int 3` instruction and handling the trap issued by these instruction appropriately, the `DosDynamicTrace` system call allows intercepting procedure calls in device drivers or the kernel. Using expressions formulated in a specialized language, this

infrastructure along with the tool *dtrace*[1] allows calls as well as local variable contents to be captured and logged.

Inspired by Dynamic Trace, *Dynamic Probes* (DProbes) implements the same approach on Linux [Moo01]. DProbes allows placing *probepoints* on arbitrary code locations in both kernel and user mode code. Like Dynamic Trace, DProbes implements probepoints by injecting an `int 3` instruction and handling traps appropriately. The replaced instruction is later either single-stepped or emulated. DProbes has also adopted the idea of using a *little language* for defining the actions to be taken whenever a probepoint becomes active.

KernInst [TM99, Tam01] is a dynamic kernel instrumentation solution for Solaris and Linux that implements runtime code splicing. It has been designed and implemented to run on unmodified kernels and can be loaded dynamically. Using splicing techniques, KernInst allows very fine-grained instrumentation – not only procedures and basic blocks, but also individual instructions can be instrumented.

On IA-32 hardware, KernInst uses injection of trap-generating instructions for implementing code splicing. Designed as a generic instrumentation solution, KernInst is capable of being used for several purposes, performance profiling and tracing being amongst them.

DTrace [CSL04] is a dynamic instrumentation solution that has been originally developed for Solaris but has meanwhile been ported to other operating systems as well, including Mac OS X and FreeBSD. Part of DTrace is the *Function Boundary Tracing Provider*, which allows dynamic tracing of procedures. Like KernInst, the IA-32 implementation of DTrace relies on the injection of trap generating instructions. Instrumenting a procedure works by replacing one of the first instructions of a procedure by a trap-generating instruction. For this to work, DTrace requires a procedure to begin with the common prolog `push ebp; mov ebp, esp`. When the procedure is executed, DTrace will handle the trap and trace the event. In order to continue execution, the replaced instruction is emulated and control is transferred back to the remainder of the traced routine. In a similar manner, DTrace allows exit tracing by replacing `ret`, `leave` or `pop ebp` instructions with a trap-generating instruction and handling the trap appropriately. To allow such instrumentation, DTrace relies on information obtained from the disassembly of the routine. Among the other features and providers offered by DTrace, it also defines a language for expressing the actions to be taken in case of a trace event.

Kprobes is a facility provided by the Linux kernel that allows hooking of kernel mode code [MPK06] by using traps. As such, KProbes is not a tool in itself but rather serves as the basis for other debugging or monitoring tools such as *Systemtap* [Var05]. Kprobes is part of the Linux kernel 2.6 and, like DTrace, has been designed to work reliably both on single and multiprocessor systems.

3.8 Editing Code

Definition

A tracing technique that relies on in-place modification of the original code. To intercept and trace execution at certain points, additional instrumentation code is incorporated into the original code.

In contrast to the technique of *Injecting and Handling Traps*, in-place modifications are not limited to injecting trap-generating instructions.

[1] Distinguishable only by capitalization, this tool is entirely unrelated to *DTrace*.

Discussion

Incorporation of instrumentation code is mostly performed by injecting jumps into the original code which divert execution to the instrumentation code and finally back to the original code.

Although functionally similar to the idea of *Injecting and Handling Traps*, the primary benefit of this approach is that jumps are significantly more lightweight in terms of performance than the issuing and handling of a trap.

The downside of this technique, however, is the increased danger of runtime code modification-caused hazards – a topic that will be discussed in more detail in section 4.

Implementations

The variable-length instruction set of the IA-32 architecture leads to several complications regarding code editing. On the one hand, disassembly requires significantly more attention than on a fixed-length instruction set architecture. This issue is discussed in more detail in section 4.4.7.

On the other hand, the IA-32 instruction set includes instructions as short as one byte. Instrumenting code sequences including such single byte instructions by injecting a jump instruction may turn out to problematic. The shortest jump instruction offered by this architecture occupies two bytes and may thus lead to requiring more than one instruction to be replaced. However, as will be discussed in more detail in section 4.4.3, such practice can lead to issues with respect to concurrency and preemption.

Solutions such as KernInst and DTrace circumvent these challenges by reverting to trap-based solutions on the IA-32 architecture. On SPARC, however, which is a fixed length instruction set architecture [Mic07], both KernInst and DTrace rely on embedding jump instructions into existing code. In order to allow safe runtime instrumentation, in-place code modifications are restricted to comprising a single instruction only. KernInst uses these jumps to splice basic blocks while DTrace uses the jumps to direct execution flow to trampolines, which in turn transfer control into DTrace [CSL04].

Paradyn [HMC94] is an early implementation of an instrumentation solution that uses code editing in conjunction with runtime code splicing. Targeted at collecting fine-grained performance measurements for parallel applications, Paradyn injects calls to trampolines[2] into the original code. To clear space for placing jumps to these trampolines, Paradyn relocates one or more instructions to the trampoline. Once execution flow has reached the trampoline, these relocated instructions along with the respective instrumentation code is executed, before execution is redirected back to the remainder of the original code. Moreover, Paradyn allows being attached to a running process.

Built on top of Paradyn, *Dyninst* [BH00] provides a framework and an object oriented API for controlling and instrumenting processes.

GILK [Pea00, PKFH02] is an instrumentation tool for the Linux kernel. Similar to KernInst, GILK implements runtime code splicing and allows instrumentation on basic block level. Unlike KernInst, however, GILK uses jumps rather than trap handling techniques to divert execution flow on IA-32 systems. As will be discussed in section 4.4.3, GILK only works on non-preemptable Linux kernels and does, in its current state, not support SMP systems.

[2] Paradyn uses two types of trampolines, base trampolines and mini trampolines. However, the differences between these types of trampolines are irrelevant to this discussion.

Another implementation of code editing on IA-32 is *Detours* [BH99]. Although limited to user mode code, Detours is noteworthy for providing a lightweight, yet widely applicable solution that poses only little requirements on the code to be instrumented. Detours can be used to either redirect execution to an alternative implementation of a routine or to intercept function entry. In both cases, Detours replaces one or more instructions located at the very beginning of a function in order to preprocess or redirect a call. The replaced instructions are moved to a *trampoline*, which, after preprocessing has been finished, is used to resume execution. Potentially replacing multiple instructions, Detours is, however, exposed to a number of issues that are discussed in section 4.4.

Vulcan [SEV01] is similar to GILK in that it also implements runtime code splicing. Although Vulcan is limited to user mode Windows, it stands out by the fact that it allies static and dynamic instrumentation techniques for a number of different architectures (IA-32, IA-64, MSIL) in a single tool.

Djprobe [Mas07] is an enhancement of Kprobes that uses jumps rather than traps to instrument routines. In comparison to Kprobes, the applicability of djprobe is slightly more restricted. Yet, due to the usage of jumps rather than traps, djprobe performs significantly better than Kprobes [Hir05], which is also the main motivation behind this approach.

Djprobe is also noteworthy for thoroughly addressing safety issues related to multiple instruction modification and patching of potentially non-quiescent code. A key aspect of their algorithm for preventing preemption-related hazards is to rely on `freeze_processes`. This routine, which is primarily used for system suspension, puts all user mode processes as well as all kernel threads that have registered as being *freezable* effectively to sleep [Wys08]. However, the paper does not discuss the safety of this approach in the existence of non-freezable kernel threads, i.e. threads that have not registered as being freezable but may still be affected by certain code patches.

3.9 Concluding Remarks

The tracing solutions presented as part of the discussion of tracing techniques vary in both their implementation as well as in the individual aims they pursue. A general assessment is thus not easily possible. Yet, moving the focus to function boundary tracing in the context of the Windows NT kernel and the applicability of the techniques in this regard, allows a finer analysis of the strengths and weaknesses of these approaches to be made.

Solutions *modifying the environment* generally require the least effort to implement and avoid the challenges associated with code modification. Notwithstanding their lightweightness and robustness, the applicability of these approaches is limited. The NT kernel includes several areas of potential applications [MJ07] where extensive use of function pointers is made, yet the share of functions traceable with this approach is, although hardly quantifiable, rather low.

In sharp contrast to this, solutions *interposing code execution* are extremely versatile and universally applicable. The fact that these solutions allow instrumentation on a much finer level than required for function boundary tracing should also be considered positive. As indicated before, however, the achilles heel of these solutions is the bootstrapping. To successfully interpose code execution, the tracing facility must either be present from the start of the respective program on or it must be injected and be provided some form of jump aid. However, requiring the tracing facility to be present during program startup has to be considered a contradiction to the tracing solution being truly *dynamic*. In the second case, the tracing solution can only be as good as its jump start facility is – the more limited the number of potential entry points is, the more limited will the applicability of such a solution be.

Solutions relying on *injecting and handling traps* or *editing code* share many properties. They do not reach the versatility of interposition solutions but compensate this shortcoming by usually being less intrusive and providing a larger number of potential instrumentation/entry points. Such solutions are, however, exposed to the challenges of dynamic code modification as discussed in section 4. Addressing all of these problems is vital for providing a code modifying solution that can be considered safe enough for productive use.

The main advantage of avoiding traps and to favor code editing techniques is performance. Trap handling introduces a significant overhead which has to be paid for each event. The cost of issuing and handling breakpoints has been reported to be even higher in case of the operating system running in a hypervisor such as Xen [Mas07].

4 Challenges Of Runtime Code Modification

This section discusses various challenges imposed by the CPU and operating system on runtime code modification. Unless stated otherwise, the discussion will focus on the Intel IA-32 architecture and applies to Windows NT kernel mode only.

Adopting the nomenclature suggested by the Intel processor manuals, code writing data to memory with the intent of having the same processor execute this data as code is referred to as *self-modifying code* [Int07c].

On SMP machines, it is possible for one processor to write data to memory with the intent of having a different processor execute this data as code. This process if referred to as *cross-modifying code* [Int07c].

For the remainder of this work, the act of executing self-modifying code or cross-modifying code is also referred to as *runtime code modification*.

4.1 Memory Model

In order to implement self-modifying or cross-modifying code, a program must be able to address the regions of memory containing the code to be modified. Moreover, due to memory protection mechanisms, overwriting code may not be trivially possible.

The IA-32 architecture offers three memory models – the flat, segmented and real mode memory model [Int07a]. As Windows NT relies on the flat memory model, only the flat memory model is examined in this work.

Whenever the CPU fetches code, it addresses memory relative to the segment mapped by the CS segment register. In the flat memory model, the CS segment register, which refers to the current code segment, is always set up to map to linear address 0. In the same manner, the data and stack segment registers (DS, SS) are set up to refer to linear address 0.

It is worth mentioning that AMD64 has retired the use of segmentation [Dev07] and the segment bases for code and data segment are therefore always treated as 0.

Given this setup, code can be accessed and modified on IA-32 as well as on AMD64 in the same manner as data.

4.2 Memory Protection

One of the features enabled by the use of paging on IA-32 systems is the ability to enforce memory protection. Each page can specify restrictions to which operations are allowed to be performed on memory of the respective page.

In the context of runtime code modification, memory protection is of special importance as memory containing code usually does not permit write access, but rather read and execute access only. A prospective solution thus has to provide a means to either circumvent such write protection or to temporarily grant write access to the required memory areas.

As other parts of the image are write-protected as well, memory protection equally applies to approaches that modify non-code parts of the image such as the Import Address Table.

Some instrumentation techniques rely on code generation. Assuming Data Execution Prevention [Cor06a] has been enabled, it is vital for such approaches to work properly that any code generated is placed into memory regions that grant execute access. While user mode implementations can rely on a feature of the RTL heap (i.e. using the `HEAP_CREATE_ENABLE_EXECUTE` when calling RtlCreateHeap) for allocating executable memory, no comparable facility for kernel mode exist – a potential instrumentation solution thus has to come up with a custom allocation strategy.

4.3 Jump distances

A tracing solution may employ dynamic generation of branching instructions in order to splice basic blocks and thus to adapt execution flow. For certain approaches, the offset between the branching instruction itself and the jump target may be of significant size. In such cases, the software has to make sure that the branch instruction chosen does in fact support offsets at least as large as required for the individual purpose.

The IA-32 instruction set offers a number of branch instructions which, among other characteristics, differ in the maximum jump distance they support. As an example, a near jump with immediate operand can address targets with an offset relative to the current instruction pointer that can be expressed by a signed 32 bit value. Short jumps, in comparison, support only 8 bit offsets.

However, as those jump instructions that support the largest distances also tend to be the largest in size, it is often advantageous to use a smaller, yet more restricting jump. In case the instrumentation solution is basically rewriting the basic block and its size is allowed to change, it may opt to replace a jump instruction by an instruction supporting a larger displacement. But in case modifications are performed in-place and the size of the basic block is therefore limited, this technique can usually not be applied.

A common technique to overcome this issue is to make use of trampolines [TM99]. To support larger jump offsets than the actual instruction supports, a stopover is introduced: Execution jumps from the origin to a trampoline, which is a snippet of code consisting of a single jump instruction only. Taking this jump, execution is then redirected to the ultimate target. If the trampoline can be located close to the origin, the injected branch instruction only has to support a very short distance.

On Windows NT, Tamches and Miller [TM99] suggest overlaying initialization code, which can be assumed not to be required any more, yet being close to the code locations being instrumented, with trampolines. However, this approach fails to take into consideration that such code regions (i.e. routines such as `DriverEntry`) are commonly located in paged memory. All accesses to these memory regions therefore have to occur at *Interrupt Request Level* (IRQL) below `DISPATCH_LEVEL` in order not to risk a bugcheck. Yet, as the IRQL at which such trampolines might be executed is hard to determine beforehand, adherence to this restriction may not be guaranteed. As such, the applicability of this approach has to be expected to be limited.

4.4 Safety Concerns

As indicated by the previous section, the act of treating code as data and performing modifications on code is straightforward. The question remaining to be solved, however, is the safety of such actions, especially in the presence of multiple processors.

Runtime code modification in general can be used for a variety of purposes and a discussion of the overall safety of such actions is out of scope of this work. Therefore, this section

concentrates on the safety in the specific context of using Runtime Code Patching with the intent of allowing routines to be traced.

Given a multiprogramming environment, an algorithm for dynamic instrumentation using runtime code modification shall be considered *safe* in the context of this work if its usage guarantees that at any point in time, a routine is either run unchanged, or is run in an *instrumented* state.

Running a routine in an instrumented state includes:

- Calling a pre-hook routine.
- Running an execution path equivalent to the execution path that would have been taken in case of running the unmodified routine.
- Calling a post-hook routine. (Optional)
- Running any necessary thunking code.

The execution path of an instrumented routine is considered equivalent to the execution path of an unmodified routine if it includes all non-noop instructions of the unmodified routine (in the same order) and any additional instructions do not affect the outcome of the routine in terms of return value, output-parameters and modified memory. Noop instructions shall include the instruction `nop` as well as any other instructions that carry out no semantically useful work such as `mov edi, edi`.

The cited criteria should serve as a practical guideline for the evaluation of instrumentation approaches in the remainder of this work. Both the discussion of whether these criteria are in fact sufficient for a solution to work correctly under all possible circumstances and proving whether or not a potential instrumentation algorithm meets these criteria is out of scope of this work. Rather, the remainder of this section will discuss common issues in the light of these criteria and their individual ramifications.

The range of potential issues discussed in the following sections is a summary of what has been discussed by the publications of existing instrumentation solutions presented in section 3 as well as issues experienced with existing tools.

Although not aiming to be a complete list of possible issues, an algorithm not being prone to any of these issues while meeting the criteria above should be able to be considered *safe for practical usage*.

It is worth noting that this section does not address any issues not applying to the IA-32 architecture but that may still be relevant on other architectures such as issues regarding out of order instructions, delay slots or non-transparent instruction caches.

4.4.1 Cross-Modifying Code and Atomicity

Performing modifications on existing code is a technique commonly encountered among instrumentation solutions. Assuming a multiprocessor machine, altering code brings up the challenge of properly synchronizing such activity among processors.

As stated in section 4.1, code can be modified in the same manner as data. Whether modifying data is an atomic operation or not, depends on the size of the operand. If the total number of bytes to be modified is less than 8 and the target address adheres to certain alignment requirements, current IA-32 processors guarantee atomicity of the write operation [Int07c].

If any of these requirements do not hold, multiple write instructions have to be performed, which is an inherently non-atomic process. It is, however, crucial to notice that even in situations where using atomic writes or bus locking on IA-32 or AMD64 would be feasible, such practice would not necessarily be safe as instruction fetches are allowed to pass locked instructions [Int07c].

Although appealing, merely relying on the atomicity of store operations must therefore in many cases be assumed to be insufficient for ensuring safe operation.

The exact behavior in case of runtime code modifications slightly varies among different CPU models. On the one hand, guarantees concerning safety of such practices have been lessened over the evolvement from the Intel 486 series to the current Core 2 series [Int07c]. On the other hand, certain steppings of CPU models exhibit defective behavior in this regard [Int02].

Due to this variance, the exact range of issues that can arise when performing code modifications is not clear and appropriate countermeasures cannot be easily identified. As described in several errata, [Int02], cross-modifying code not adhering to certain coding practices described later, can lead to *unexpected execution behavior*, which may include the generation of exceptions.

The route chosen by the Intel documentation is thus to specify an algorithm that is guaranteed to work across all processor models – although for some processors, it might be more restricting than necessary [Int07c].

For cross-modifying code, the suggested algorithm makes use of *serializing instructions*. The role of these instructions, `cpuid` being one of them, is to force any modifications to registers, memory and flags to be completed and to drain all buffered writes to memory before the next instruction is fetched and executed [Int07c].

Quoting the algorithm defined by Intel in [Int07c]:

```
(* Action of Modifying Processor *)
Memory_Flag <- 0; (* Set Memory_Flag to value other than 1 *)
Store modified code (as data) into code segment;
Memory_Flag <- 1;

(* Action of Executing Processor *)
WHILE (Memory_Flag != 1)
  Wait for code to update;
ELIHW;

Execute serializing instruction;
Begin executing modified code;
```

To further complicate matters, the IA-32 architecture uses a variable-length instruction set. As a consequence of that, additional problems not yet addressed may occur if the instruction lengths of unmodified and new instruction do not match. Two situations may occur

- The new instruction is longer than the old instruction. In this case, more than one instruction has to be modified. Modifications straddling instruction boundaries, however, are exposed to an extended set of issues discussed in section 4.4.3.

- The new instruction is shorter than the old instruction. The ramifications of this situation depend on the nature of the new instruction. If, for instance, the instruction is an unconditional branch instruction, the subsequent pad bytes will never be executed and can be neglected. If, on the other hand, execution may be resumed at the instruction following the new instruction, the pad bytes must constitute valid instructions. For this purpose, a *sled* consisting of `nop` instructions can be used to fill the pad bytes.

  The algorithm defined by Intel for cross-modifying code ensures that neither the old nor the new instruction is currently being executed while the modification is still in progress. Therefore, when employing this algorithm, replacing a single instruction by more than one instruction can be considered to be equally safe to replacing an instruction by an equally-sized instruction.

It is worthwhile to notice that regardless which situation applies for instrumentation, the complementary situation will apply to uninstrumentation.

4.4.2 Concurrent Execution

A typical user mode process on a Windows system can be expected to have more than one thread. In addition to user threads, the Windows kernel employs a number of system threads. Given the presence of multiple threads, it is likely that whenever a code modification is performed, more than one thread is affected, i.e. more than one thread is sooner or later going to execute the modified code sequence.

The basic requirement that has to be met is that even in the presence of a preemptive, multi threaded, multiprocessing environment, an instrumentation solution has to ensure that any other thread either does not run the affected code at all, runs code not yet reflecting the respective modifications or runs code reflecting the entire set of respective modifications.

On a multiprocessor system, threads are subject to concurrent execution. While one thread is currently performing code modifications, another thread, running on a different processor, may concurrently execute the affected code.

If only a single instruction is to be modified and the cited algorithm for cross-modifying code is used, concurrent execution, preemption and interruption should not be of concern. Any other thread will either execute the old or new instruction, but never a mixture of both.

However, the situation is different when more than one instruction is to be modified. In this case, a different thread may execute partially modified code.

Although code analysis may indicate certain threads not to ever call the routine comprising the affected code, signals or *Asynchronous Procedure Calls* (APCs) executed on this thread may. Therefore, a separation in affected and non-affected threads may not always be possible and it is safe to assume that all threads are potentially affected.

4.4.3 Preemption and Interruption

Both on a multiprocessor and a uniprocessor system, all threads running in user mode as well as threads running in kernel mode at IRQL APC_LEVEL or below are subject to preemption. Similarly, for a thread running at DISPATCH_LEVEL or *Device IRQL* (DIRQL), it is also possible to be interrupted by a device interrupt [SR04]. As these situations are similar, only the case of preemption is discussed.

If only a single instruction is to be modified, preemption and interruption may not be problematic. If, however, multiple instructions are to be adapted, the ramifications of

preemption in this context are twofold. On the one hand, the code performing the modification may be preempted while being in the midst of a multi-step runtime code modification operation:

- Thread A performs a runtime code modification. Before the last instruction has been fully modified, the thread is preempted. The instruction stream is now in a partially-modified state.
- Thread B begins executing the code that has been modified by Thread A. In case instruction boundaries of old and new code match, the instruction sequence that is now run by Thread B should consist of valid instructions only, yet the mixture of old and new code may define unintended behavior. If instruction lengths do not match, the situation is worse. After Thread B has executed the last fully-modified instruction, the CPU will encounter a partially-overwritten instruction. Not being aware of this shift of instruction boundaries, the CPU will interpret the following bytes as instruction stream, which may or may not consist of valid instructions. As the code now executed has never been intended to be executed, the behavior of Thread B may now be considered arbitrary.

In order to avoid such a situation from occurring, an implementation can disable preemption and interruption by raising the IRQL above DIRQL during the modification process.

On the other hand, the code performing the code modification may run uninterrupted, yet one of the preempted threads might become affected:

- Thread A has begun executing code being part of the instruction sequence that is about to be modified. Before having completed executing the last instruction of this sequence, it is preempted.
- Thread B is scheduled for execution and performs the runtime code modification. Not before all instructions have been fully modified, it is preempted.
- Thread A is resumed. Two situations may now occur – either the memory pointed to by the program counter still defines the start of a new instruction or – due to instruction boundaries having moved – it points into the middle of an instruction. In the first case, a mixture of old and new code is executed. In the latter case, the memory is reinterpreted as instruction stream. In both cases, the thread is likely to exhibit unintended behavior.

One approach of handling such situations is to prevent them from occurring by adapting the scheduling subsystem of the kernel. However, supporting kernel preemption is a key characteristic of the Windows NT scheduler – removing the ability to preempt kernel threads thus hardly seems like an auspicious approach. Regarding the Linux kernel, however, it is worth noting that kernel preemption is in fact an optional feature supported on more recent versions (2.6.x) only [BC05]. As a consequence, for older versions or kernels not using this option, the situation as described in the previous paragraph cannot occur. GILK [Pea00], which has been mentioned before for being an instrumentation tool relying on being able to replace multiple instructions in fact relies on a kernel not supporting kernel preemption.

A more lightweight approach to this problem relies on analysis of concurrently running as well as preempted threads. That is, the program counters of all threads are inspected for pointing to regions that are about to be affected by the code modification. If this is the case, the code modification is deemed unsafe and is aborted. Needless to say, it is crucial

that all threads are either preempted or paused during this analysis as well as during the code modification itself. As the thread performing the checks and modifications is excluded from being paused and analyzed, it has to be assured that this thread itself is not in danger of interfering with the code modification.

In a similar manner, the return addresses of the stack frames of each stack can be inspected for pointing to such code regions. Stack walking, however, is exposed to a separate set of issues that are discussed in section 4.4.5.

Rather than aborting the operation in case one of the threads is found to be negatively affected by the pending code modification, a related approach is to attempt to fix the situation. That is, the program counters of the affected threads are updated so that they can resume properly.

One example for a user-mode solution implementing this approach is Detours [BH99]. Before conducting any code modification, Detours suspends all threads a user has specified as being potentially affected by this operation. After having completed all code modifications, all suspended threads are inspected and their program counters are adapted if necessary. Not before this step has completed, the threads are resumed.

4.4.4 Basic Block Boundaries

Another issue of multiple instruction modification is related to program flow. Whenever a sequence of instructions that is to be altered spans multiple basic blocks [AVAU88], it is possible that not only the first instruction of the sequence, but also one of the subsequent instructions may be a branch target. When instruction boundaries are not preserved by the code modification step, the branch target might fall into in the midst of one of the new instructions. Again, such a situation is likely to lead to unintended program behavior [TM99].

Identifying basic blocks and thus any potential branch targets requires flow analysis. However, especially in the case of optimized builds, it is insufficient to perform an analysis of the affected routine only as blocks might be shared among more than one routine. In such cases, a routine does not consist of a contiguous region of code but may be scattered throughout the image. Therefore, it is crucial to perform flow analysis on the entire image. But even in this case, the existence of indirect branches may render a complete analysis impossible in practice.

The idea of creating a control flow graph in order to avoid overwriting a basic block boundary has been implemented by several solutions, including GILK [Pea00] and KernInst [Tam01], which analyzes the entire kernel image. In a similar manner, djprobe [Mas07] performs static code analysis to detect such issues. Both instrumentation tools rely on overwriting multiple instructions and are therefore exposed to this issue.

Another situation where an instrumentation solution may run into the danger of overwriting basic block boundaries is the instrumentation of very short routines. If the routine is shorter (in terms of instruction bytes occupied) than the instructions that need to be injected in order to instrument the routine, the first basic block(s) of the subsequent routine may be overwritten.

4.4.5 Stack Walking

As indicated before, walking the stacks of all threads is a technique that can be employed to check whether certain code modification operations can be expected to be safe or not. However, for such checks to be reliable, it has to be assured that the stack traces are proper, i.e. that no frames are missed.

If correct debugging symbol information is available for all modules involved in a stack walk, a stack trace can be expected to be reliable. In practice, however, debugging information is often not available, so that symbol-less approaches have to be taken.

While reliably obtaining proper stack traces without using debugging symbols may in fact be viable on certain architectures and platforms, including Windows NT/AMD64 with its unified calling convention, it turns out to be rather problematic on the IA-32 platform. To walk the stack without debugging symbols, the frame pointer chain can be attempted to be traversed. This approach, however, is thwarted as soon as one of the modules involved makes use of certain compiler optimizations such as *frame pointer omission*. As a consequence, this technique can usually not be expected to yield reliable results.

The dynamic updating solutions Ksplice [Arn08] and Dynamos therefore implement a more conservative, debugging symbol-less approach of walking the stack: Starting from the top of the stack, each double word (on IA-32) is inspected until the base address of the stack has been reached. Whenever a double word contains a value that, interpreted as a pointer, refers to a region in memory occupied by code, this address is assumed to be a return address of a stack frame. Although this assumption may be faulty and is likely to lead to false positives, this technique gives these solutions the ability to perform certain checks on the routines these addresses point to.

Although the latter approach seems auspicious, there is another inherent problem related to the overall approach of analyzing the stacks of all threads. For such analysis to be reliable, it is of utmost importance to analyze *all* stacks. In particular, this includes the stacks of all preempted threads. In the normal case, enumerating this list is possible by walking the list of threads maintained by the kernel. In user mode Windows NT, such enumeration can be done using the *Toolhelp API*, in kernel mode, the list of KTHREAD structures can be traversed.

However, this approach neglects the fact that in addition to each kernel-allocated thread being associated a stack, there may be additional, custom allocated stacks. *Fibers*, for instance, as offered by Windows NT in user mode, offer such a facility. Each fiber maintains a separate stack. Yet, as there is no 1:1 mapping between these stacks and kernel-allocated stacks, it is not possible to enumerate and identify such user-allocated stacks by using the thread-enumeration facilities discussed above. In fact, locating user-allocated stacks must be expected to not be possible without the individual program cooperating.

In NT kernel mode, the practice of maintaining user-allocated stacks may not be very common. Moreover, it is explicitly discouraged by Microsoft [Cor07]. Still, the possibility of drivers using such techniques cannot be ruled out.

Regarding stack walking, custom allocated stacks pose a serious concern. As they are equally subject to preemption-related issues as kernel-allocated stacks are, it is crucial for such stacks to be analyzed as well.

4.4.6 Life Cycle Management of Dynamically Allocated Code

A common technique encountered among dynamic instrumentation solutions is using *trampolines*, i.e. little chunks of dynamically generated code that is mostly specific to a single instrumentation point. Such code blocks, but also other code that has been dynamically loaded, is subject to proper life cycle management.

As soon as an instrumentation point is revoked and the affected routine is reverted to its original state, code blocks specific to this instrumentation point are not needed any more. However, due to the existence of concurrently running as well as preempted threads, it is

not trivial to judge whether unloading and freeing the affected memory is indeed a safe operation. For concurrency, another CPU may currently be running the code regions in question. Even more likely is the case that one of the preempted threads is referring to instructions of the affected code chunk. Depending on the individual algorithm and the question whether or not the code block may contain call instructions, the program counter of the preempted thread as well as the return addresses of any of the thread's stack frames may still point to the code block in question. Unloading and freeing the trampoline would in this case inevitably lead to faults or otherwise unintended program behavior.

Detours [BH99] attempts to avoid such situations by analyzing the state of other threads. After suspending all affected threads – the distinction between affected and non-affected threads has to be made by the user of the library – Detours checks whether any thread refers to the trampoline that is to be freed. If this is the case, the instruction pointer of this thread is modified to point to the original code again, which will have been restored before the suspended threads are resumed.

However, by inspecting the current instruction pointer (i.e. the top stack frame) of the threads only, Detours fails to consider the possibility of lower stack frames being affected: Under certain circumstances, it is possible that Detours will relocate a call instruction into the trampoline. In this case, the following situation may occur:

- Thread A enters the trampoline, executes the call and enters the called routine.
- Thread B removes the detour. This includes freeing the respective trampoline.
- Thread A returns from the called routine and will attempt to continue execution in the trampoline. As the trampoline has already been freed, an access violation or undefined behavior will occur.

MDL [HMG+97], part of the Paradyn tools, therefore analyzes all stack frames before freeing a trampoline. In case an affected stack frame has been identified, the free operation is delayed and retried later. This analysis is significantly more thorough and avoids situations such as the one Detours suffers from. Yet, the viability of this approach depends on whether it is possible to always perform a proper stack walk of the affected threads – a topic that has been discussed in section 4.4.5.

KernInst [TM99], which makes extensive use of dynamically allocated code blocks for trampolines and code patches, attempts to avoid lifecycle problems from occurring by delaying free operations. Solaris, like Windows NT, both supports SMP and kernel thread preemption. Short of being able to identify whether a preempted thread or concurrently running thread is still referring to the affected code, KernInst does not immediately free the memory when the instrumentation is revoked. Rather, it waits three seconds, during which it expects all affected threads to have left the critical region before finally freeing the memory. This expectation is backed by the fact that all operations performed within trampolines are non-blocking. Yet, this strategy is only able to minimize rather than to eliminate the likeliness of freeing code blocks that are still in use.

Attempts to solve this issue by using reference counting, as suggested by [Tam01], are exposed to a hen-egg problem: When entering a dynamically allocated code block, a counter-incrementing instruction is executed, signaling that the block is in use. Before leaving the block, the counter is decremented. As soon as the associated instrumentation has been revoked and the counter drops to zero, the block may be freed. The problem of this approach lies in the fact that decrementing the counter and taking the jump to leave the block cannot be performed in a single, atomic operation. The affected thread may successfully have decremented the counter to zero, but is preempted before having

been able to take the jump to leave the block. Referring to the reference counter, the block is now in a state that allows it to be freed – yet, a thread is still referring to it. Again, additionally delaying the free operation can help minimizing, but is unable to eliminate this risk.

In a similar manner, dynamically allocated code shared among several or all instrumentation points is affected by this discussion. For instance, when the instrumentation solution has been developed as a loadable driver, it may not always be easily possible to determine whether unloading this driver would be a safe operation. Although any instrumentation may have long been revoked, some preempted (possibly starved) thread may still be referring to a routine located within the driver that has been called by instrumentation code.

4.4.7 Disassembly

Certain instrumentation algorithms require partial or even complete disassembly of the routine to be instrumented. However, the relative complexity of the IA-32 instruction set, the fact that instructions are of variable length and the problem that code and data can often not be clearly separated from each other makes decoding IA-32 machine code nontrivial. Moreover, the existence of indirect jumps can thwart attempts to perform thorough static analysis on the code.

Faced with certain problematic code and data sequences, both of the two predominant disassembly algorithms, *linear sweep* and *recursive traversal* can run into situations where they silently begin to produce wrong results. Although the results have been shown to be improvable by hybrid approaches, the general problem remains in existence [SDA02].

Regarding compiler-generated code, such problematic code sequences rarely occur. Rather, such code is often a sign of deliberate code obfuscation [Eil05]. Still, in the context of reengineering-sensitive software, which not only malware but also systems like *Digital Rights Management* (DRM) solutions belong to, such code obfuscation is quite popular and should not be neglected in practice.

The nature of a dynamic tracing solution implies that it must be capable of dealing with unknown code, i.e. code that has never been instrumented before with the given solution. In order to properly cope with such code, an instrumentation solution relying on disassembly and static analysis should therefore put emphasis on the quality of disassembly as it may directly influence the stability of the system.

4.4.8 Parameter Validation

Although not strictly related to the practice of instrumentation, an important factor of the safety of instrumentation is proper usage and validation of parameters. Whenever an instrumentation solution has intercepted a routine call and wishes to access any of the parameters passed to the routine, it must be capable of dealing with invalid parameters. The situation where proper parameter validation is especially crucial for system stability is when any of the parameters contain values specified by a user mode program. As this is the case for all system service routines, intercepting these routines and using the parameters passed requires special care. Practices such as using a pointer without proper validation or dereferencing an object handle without type checking can undermine the security and stability of the operating system [Joh06].

4.4.9 Other Events

Further events an instrumentation solution may not have control over may influence proper completion of a runtime code modification. An example of such an event are DMA operations. A device – without the instrumentation software being aware of – might start a data transfer affecting the memory location that is currently affected by a code modification. Due to concurrent accesses, the result of such an operation may thus become undefined.

4.5 Sharing of Resources

The vast majority of tracing solutions can be expected to run parts or even most of their code in the same virtual address space as the tracee. This holds true for code editing-based systems but may certainly also apply to other instrumentation solutions which utilize different tracing techniques.

As such, the tracing code and the traced code share a number of resources, including memory, registers, condition flags, and handles.

While the tracing code is aware of potential conflicts that may arise from this sharing, the traced code is not. As such, it is crucial to share these resources in a manner that avoids conflicts and impacts the traced code to the least amount.

Cmelik [CK94] defines the following four ways to deal with such sharing of resources:

- Partition the resource. Part of the resource belongs to one contender, and part to another. This, for example, is applicable to heap/pool memory.
- Time multiplex the resource. Part of the time the resource belongs to one contender and part of the time it belongs to another. Additional processing time and storage are required for swapping. Use cases for this approach include sharing registers between the contenders.
- Simulate (virtualize) the resource to the tracee.
- Unprotected sharing – changes made by either effect both. This is generally to be avoided.

There are ample resources whose sharing between the traced and tracing party may lead to problems. However, of special interest are those resources that are modified as part of the instrumentation process. Depending on the tracing technique, this may be code, function pointers, or other resources. Such modifications are conducted to influence the execution of the tracee. Yet, running in the same address space, any of such modifications may also directly or indirectly influence the tracing code, which is generally undesired.

One of the most prominent issue in this regard is reentrance. For example, a routine like `malloc` may have been instrumented and is currently traced. Yet, for tracing to work, a memory allocation may have to be performed on behalf of the tracing code. However, calling `malloc` in this situation will invoke the tracing code again, which in turn will lead to endless recursion.

But even if such recursion is avoided, reentrance may become a problem. The implementation of `malloc` in the previous example may not be reentrant or – worse yet – may deadlock in case of reentrant calls. Again, the situation depicted may well lead to a crash or a hang.

Code modification, even if conducted safely, may lead to undesired, resource sharing-related effects as well: For example, the instrumentation process may involve injecting `int 3` instructions into the the code, with the intent of handling the traps appropriately. However, if the instrumented program is being debugged, this practice may well interfere with the debugger's notion of breakpoints, which may also rely on `int 3` instructions being injected. Clearly, such effects would be a result of the interrupt being shared among both parties without an ordered sharing strategy.

Properly dealing with shared resources and potential effects such as reentrance is therefore of utmost importance for the proper working of a tracing solution.

4.6 Evaluation of Safety Concerns

Having discussed prevalent challenges of runtime code modification certain conclusions may be drawn. An important observation is that due to the inherent complexities and challenges coming along with replacing multiple instructions, an approach that only requires replacing a single instruction of non-quiescent code is clearly favorable.

Moreover, not all issues are equally critical. Some events such as concurring DMA operations on memory containing the code to be modified are unlikely and uncommon enough to be neglected in practice.

Part II

NTrace

5 Architectural Overview

Before discussing the approach and implementation of NTrace in detail, the overall architecture of the system shall be briefly presented. The system has been decomposed into a number of components in order to allow for the ability to reuse certain components by other tools. Figure 5.1 shows an architectural overview.

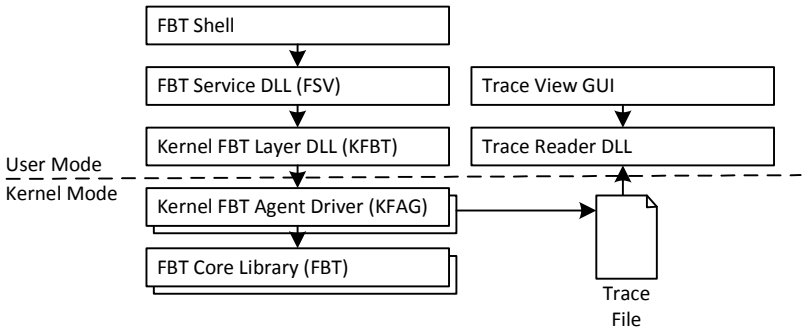

Figure 5.1: Buffer Management Dynamics

The actual tracing mechanics have been implemented as a static library, the *Function Boundary Tracing (FBT) Core Library*. Both a user mode and a kernel mode version of this library have been implemented, which share most parts of their code. Moreover, to account for certain differences in their implementation, two kernel versions of this library exist – one for the *Windows Research Kernel* (WRK) [PP06] and one for retail kernels.

The FBT Core Library performs the bulk of instrumentation and tracing, yet it is not workable on its own – certain aspects, such as event handling, are not handled by this library itself. The FBT Core Library is embedded into the *FBT Agent*, which has been implemented as a device driver. Providing an IOCTL-based interface, this driver wraps the functionality of the FBT Core Library and implements event handling by maintaining a trace file events are asynchronously written to. Again, two editions of this driver exist – one for the WRK and one for retail editions of Windows.

Besides linking against different editions of the FBT Core Library, the WRK version has additionally been augmented by the capability of working in conjunction with the *Windows Monitoring Kernel* (WMK) [SS07]. The WMK is an enhancement of the WRK, specialized for the purpose of fine-grained monitoring. Besides providing the ability to trace a variety of events including wait events, system calls and context switches, it provides a dedicated tracing API. This API can optionally be used by the WRK version of the Kernel Function Boundary Tracing Agent, provided it is run on a capable kernel.

The *FBT Layer DLL* is a user mode component that simplifies the usage of the IOCTL-based interface of the Agent. Moreover, it handles selection, lazy installation and dynamic loading of the appropriate driver. This library, as well as all kernel mode components, does not make use of symbols but expects the user to work with raw function virtual addresses when requesting functions to be instrumented. All symbol handling is performed uniformly by the *Function Boundary Tracing (FBT) Service DLL*.

The *FBT Service DLL* constitutes the top level library. Its main duties include implementing symbol management and maintaining certain bookkeeping information such as the list of

active instrumentations. As such, this DLL makes extensive use of dbghelp [Cor08a], a Microsoft provided library for symbol management. On top of this functionality, it also implements a simple command interpreter.

This command interpreter is utilized by the FBT Shell, a console based application. With the help of the FBT Service DLL, it provides a user experience similar to the cdb/ntsd debuggers [Cor08b], although the set of available commands is strictly limited to tracing-related tasks. Table 5.1 lists a subset of the commands offered.

Table 5.1: A subset of the commands offered by the FBT shell

Command	Description
tc	Revoke instrumentation on one or more routines
tl	List active *tracepoints*, i.e. traced routines
tp	Set one or more tracepoints. Example: `tp nt!Ke*`
x	Search symbol

Finally, a separate *Trace Reader* library has been implemented for reading the trace file, which employs a custom binary format. Leveraging memory mapped file techniques, the aim of this library is to provide efficient access to the trace information and to compensate for the possibility of lost events. Based on this library, a simple GUI application, *Trace View*, allows trace information to be inspected. Utilizing a Tree View control, it also visualizes the nesting of calls.

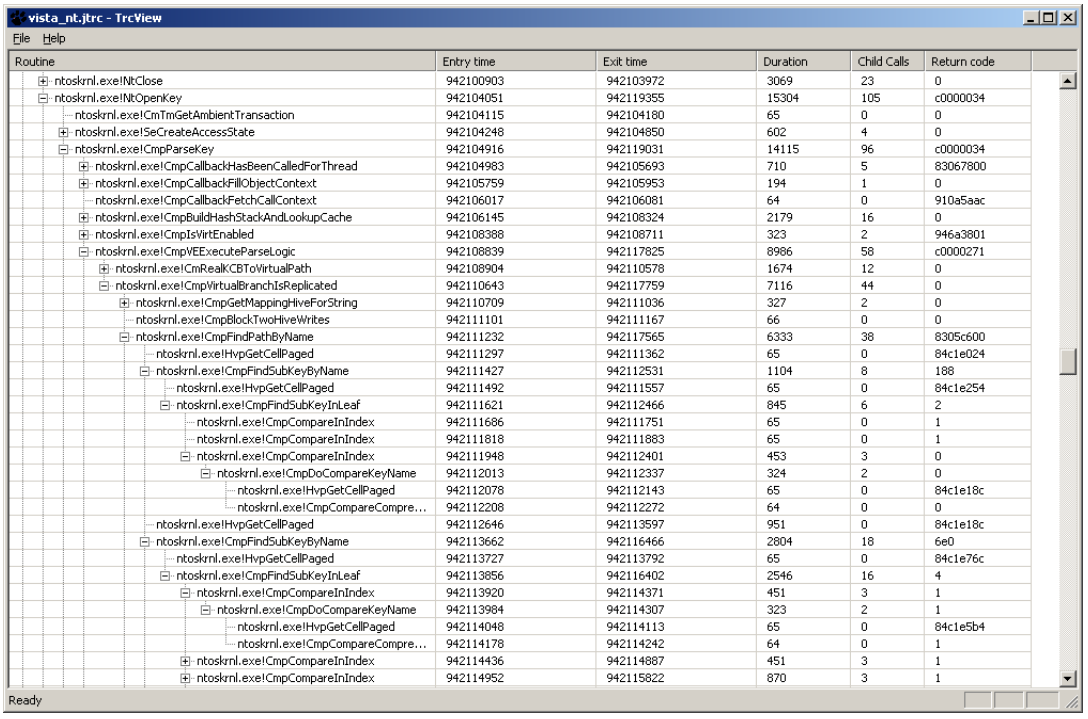

Figure 5.2: Screenshot of the Trace View application, showing a trace recorded on Windows Vista

6 Approach

6.1 Context

Based on the classification work discussed in chapter 3, *Editing Code* has been chosen as the technique to base NTrace on. The technique promises to be both well-suited for implementing function boundary tracing as well as allowing tracing to be implemented in a fashion that delivers sound performance. Yet, a prerequisite for successfully applying this technique clearly is to properly address the runtime code modification concerns discussed in chapter 4.

Focusing on the Windows NT platform, an important observation in this regard has been that to achieve safe runtime code modification, certain aspects of the Microsoft Hotpatching Infrastructure can be leveraged. To illuminate this synergy, the respective parts of the Hotpatching Infrastructure shall be briefly discussed.

The aim of the Hotpatching Infrastructure, which has been described and is covered by a patent held by Microsoft [Gar02] is to provide a means for updating binary modules during runtime by *replacing* faulty routines by new, updated routines. In practice, this replacement is achieved by having old routines redirect execution to the new, updated routines. To install such redirections, Hotpatching, as its name suggests, employs a certain amount of runtime code modification.

The key aspect Hotpatching relies on in order to make such redirections safe in terms of runtime code modification, is the notion of a *hotpatchable image*. Hotpatchable images, which have been introduced along with the Hotpatching Infrastructure in Windows Server 2003 SP1, are characterized by being built using dedicated compiler and linker switches, namely */hotpatch* and */functionpadmin* respectively:

- The compiler flag */hotpatch* effects the first instruction of a routine to be a `mov edi, edi`. This instruction is essentially a noop and occupies 2 bytes. These two properties – that it does not have any influence on the routine's semantics and its length are crucial for the further discussion.

  For certain routines, especially for routines consisting of few instructions only, however, the compiler is free to neglect this switch and in fact does not emit said instruction. Notwithstanding this limitation, experience shows that the fraction of such routines is reasonably small.

- The linker flag */functionpadmin* effects that each routine is preceded by a certain amount of padding. The size of the padding can be specified as an argument. Empirical analysis reveals that the linker usually fills this padding area with `int 3` (opcode 0xCC) or `nop` (opcode 0x90) instructions.

  Unless stated otherwise, the remaining discussion will imply the padding size to be five bytes.

While the exact mechanics of how the Hotpatching Infrastructure utilizes these properties of hotpatchable images are only partly documented, their characteristics will play a fundamental role in the further discussion of NTrace.

6.1.1 Build Environments

With hotpatchable images and the usage of the respective compiler and linker switches becoming a prerequisite for successful instrumentation, it is worth regarding their support among current build environments in more detail.

Although both switches have meanwhile been documented and are officially supported, it turns out that no clear information about the availability among compiler and linker versions seems to be currently available. A non-authorative list of compiler and linker versions along with their capabilities with respect to these switches is shown below. The value *(yes)* (in parentheses) means that the flags are undocumented, yet workable.

Table 6.1: Support of /hotpatch and /functionpadmin among compiler versions

Product	*cl* version	*link* version	Supported?
Visual Studio 2003 SP1	13.10.6030	7.10.6030	(yes)
Visual Studio 2005	14.00.50727.42	8.00.50727.42	yes
Visual Studio 2005 SP1	14.00.50727.762	8.00.50727.762	yes
DDK 3790	13.10.2179	7.10.2179	no
DDK 3790.1830 (SP1)	13.10.4035	7.10.4035	(yes)
WDK 6000	14.00.50727.220	8.00.50727.220	yes

Although their compilers support the respective switches, Visual Studio 2003 and 2005 do not advertise the use of these features, i.e. they are not exposed to the configuration interface and are not used by default. As a consequence, most modules built with Visual Studio may be expected to be not hotpatchable.

For kernel mode code, which NTrace clearly focuses on, the situation is different. As of DDK 3790.1830, the default driver build environment (i.e. makefiles) Microsoft encourages customers to use employs both switches. Moreover, empirical analysis of the respective modules reveals that current kernels and drivers provided by Microsoft with Windows Server 2003 SP1 and newer releases such as afd.sys, ntfs.sys, or tcpip.sys but also user mode libraries such as kernel32.dll and user32.dll and even applications such as notepad.exe all have been built using the respective compiler and linker-switches.

6.1.2 Operating System Releases

As indicated, Windows Server 2003 SP1 has been the first release to introduce hotpatchable images. As a consequence, the further discussion will refer to this and later releases only.

The WRK plays an another important role in the discussion of the proposed dynamic tracing solution. The WRK comprises the source code of the Microsoft Windows XP x64/Server 2003 SP1 kernel[1] and has been made available for academic purposes. As such, the WRK not only allows study of the kernel sources, it also allows modifications to be applied and customized kernel images to be built.

Initially, the applicability of the tracing solution was limited to a customized version of the WRK that included a small number of code changes. By applying certain modifications, however, it was possible to port the entire system to the retail versions of Windows Server 2003 SP1/SP2 and Windows Vista. Still, the availability of source code simplified the development.

Regarding the build settings of the WRK, it is worth pointing out that they do not seem to fully reflect the build settings used by the retail kernel. Although the /functionpadmin:5

[1] Certain parts of the source code, most notably, the implementation of hotpatching, have been omitted. However, as Microsoft provides binary objects for these parts, it is still possible to build the kernel.

linker switch is used during building, the /hotpatch switch is not. As a consequence, a WRK kernel built with the default settings will not be hotpatchable. This stands in contrast to the retail kernel, which seems to have been built with both respective switches used as it is in fact largely hotpatchable. However, this limitation of the WRK is easily overcome by adding the /hotpatch switch in the appropriate makefile and rebuilding the kernel.

6.1.3 Symbol Management

Not only targeting the WRK itself but also the retail kernel and drivers has another implication on symbol management.

An essential requirement for a dynamic tracing solution is to be able to locate routines within a loaded binary. As the code itself does not provide sufficient information to accomplish this, additional sources of information have to be used. The two major sources of such information are *map files* and *debugging symbols*, both of which can be generated during the build of a module. Map files are not well suited for automatic consumption, so the discussion will focus on debugging symbols, which are specially tailored to this purpose.

The current debugging infrastructure on Windows requires compilers not to embed this debugging information into the module itself but store it in a separate file, the *program database* (PDB). If a PDB contains the entire set of debugging symbols, which is the default, it is said to provide *private symbols*. For the binaries of Windows and other applications, Microsoft does not provide private symbols but rather a stripped-down version of PDB files only, which are said to provide *public symbols*. Public symbols lack most of the detail information provided by private symbols.

With regard to functions, private symbols include very detailed information such as name, offset, parameter information and calling convention. In contrast, public symbols merely provide the name and the offset. Moreover, distinguishing between functions and global variable names is not possible any more.

While a tracing solution targeting the WRK exclusively could in fact rely on private debugging information, this requirement is prohibitive when targeting retail editions of the kernel and drivers. Therefore, one requirement for NTrace was to restrict symbol usage to public symbols.

6.2 Operation

Having discussed the context, the basic operation of the function boundary tracing approach implemented can now be described. The following sections provide a walk through of how trace events are captured and consumed. Yet, most implementation details will be ignored at this stage and will be discussed in chapter 7.

6.2.1 Function Entry Tracing

Function boundary tracing includes both tracing entry and exit of a given function. Yet, being able to trace function entry is the more basic requirement of the two and shall be discussed first.

To leverage the properties of hotpatchable images, the basic functionality of function entry tracing is similar to the idea of hotpatching discussed before. To illustrate this functionality, Figure 6.1 shows an example of a function `Foo` that is to be traced:

Execution reaches `Foo` (1), which has been instrumented before. Part of this instrumentation is that the first instruction, which has to be a `mov edi, edi`, has been replaced with a branch to `Foo-5`. Following this jump (2), execution reaches the padding area mentioned

before which precedes every hotpatchable routine. This area has been initialized with a trampoline, which basically comprises a single instruction that directs execution (3) to a special routine (tentatively named `CaptureEntryEvent`) which traces the call. This routine is part of the tracing framework. After the tracing information has been obtained, execution is redirected back (4) to `Foo`. Yet, to avoid an infinite loop, execution continues at the second instruction, i.e. the instruction following the `mov edi, edi` (which has meanwhile been replaced by the branch instruction). Following on, `Foo` completes as normal and will finally return to the caller (5).

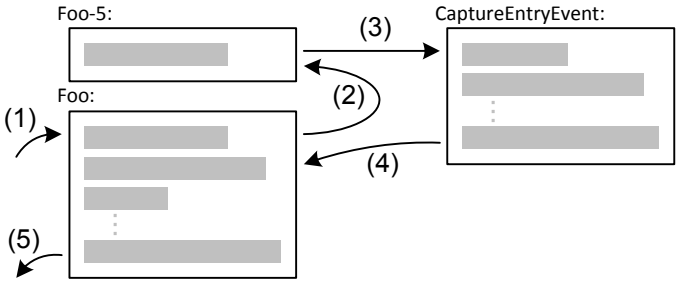

Figure 6.1: Schematic execution flow for tracing entry events of a function Foo

6.2.2 Function Exit Tracing

As discussed so far, only function entry events can be captured. To trace function exit, the tracing solution has to regain control before the traced routine will return to the caller.

To implement this regain of control, the function's return address is modified. This idea corresponds to what has been described before by Brown [Bro99b] in the discussion of the COM Universal Delegator.

The return address of the traced function, located on the stack, contains the address within the calling routine at which execution is to be resumed after return from the function. As such, it points to the instruction following the call instruction that led to the execution of the respective function. The layout of a stack frame during the execution of a routine is illustrated in figure 6.2. In accordance to the IA-32 architecture, the stack is drawn as growing from top to bottom.

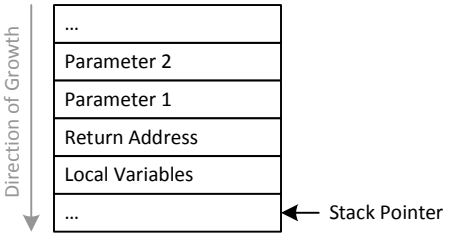

Figure 6.2: Layout of the stack during execution of a routine.

Clearly, replacing the return address is a way to regain control after the traced function has returned. However, this technique raises two questions. On the one hand, replacing

this address is not possible before the call has been initiated and the address has been pushed onto the stack. However, the function entry tracing technique discussed in the previous section, already provides a solution to this issue: As this code runs after the call instruction pointing to the traced routine has been executed, the return address is available on the stack and can be modified appropriately.

On the other hand, the question of how to preserve the original return address remains to be solved. Preserving this address for the duration of executing the traced function is crucial as this address is the only piece of information indicating where execution is to resume after completion of post-processing activities.

The storage location of the original return address not only has to be thread-local in order to be able to trace concurrently. It also has to be specific to a stack frame in order to allow tracing recursive functions or multiple functions that call each other.

The ideal place for storing the original return address would thus be the stack itself – the stack is thread-local and would also nicely allow storing the return addresses in case of recursion. Unfortunately however, usage of the stack is not possible: Pushing the address onto the stack during function entry would shift the location of local variables one machine word down the stack. While this in itself may be harmless, the offset from the stack pointer (and base pointer, if used) to the parameters increases by one machine word as well. This increase, however, will wreak havoc on the proper operation of the routine. As the assumptions about the relative location of its parameters on the stack will now be wrong, the outcome of this routine will be undefined.

To overcome this problem, the entire block of parameters could be replicated. That is, after pushing information such as the original return address onto the stack, all parameters are pushed again. This additional call frame would then have to be torn down during function exit processing. However, this approach requires knowledge about the number of parameters the individual function takes. Yet, with the restriction on public symbols, this information is not available, which in turn makes this approach infeasible.

The route chosen is thus to provide a separate, thread-local *auxiliary stack* for the storage of these return addresses. The implementation of this stack will be discussed in more detail in section 7.2.

6.2.3 Event Callbacks

Being able to gain control before function entry and after function exit puts the tracing solution into the situation where it is able to produce and trace appropriate events.

A variety of different approaches for handling such events exist. Yet, in order to keep the core tracing implementation free of a specific choice in this regard, event handling is left to other parts of the system. As a consequence, the core implementation restricts itself to calling an appropriate entry or exit callback routine that has been installed for this purpose before. This callback routine is provided not only the address of the routine that is currently being traced but also a snapshot of the general purpose register contents. It is left to the implementation of this callback how to make use of this information.

6.2.4 Putting the Pieces Together

Having discussed the basic ideas of both function entry and exit tracing, these techniques can now be combined. Figure 6.3 illustrates the schematic execution flow.

As a result of instrumenting the function `Foo`, the `mov edi, edi` instruction, which is always the very first instruction of a hotpatchable routine, has been replaced with a jump

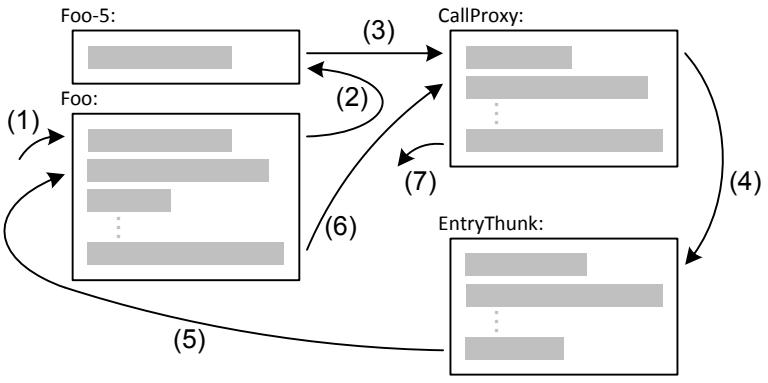

Figure 6.3: Schematic execution flow for tracing function Foo

to the padding area preceding the function. This padding area, as mentioned before, is used as a trampoline.

Using the padding area for storing the trampoline has two benefits. First, no memory has to be allocated dynamically and the lifecycle issues described in section 4.4.6 are avoided. Clearly, this benefit is bought in exchange for increased module size.

The main advantage lies in the fact that the distance between trampoline and original routine is fixed and always 5 bytes in size. This in turn allows a short jump to be used to redirect execution from the original routine to the trampoline. A short jump occupies only two bytes and is the smallest jump instruction offered by the IA-32 instruction set. This also explains the role of the `mov edi, edi` instruction – also being 2 bytes in size, the mere role of this instruction is to reserve space for placing said jump. Although this instruction introduces a certain runtime overhead for each routine, the performance impact can be expected to be negligible in most cases.

It is worth pointing out that as only the initial `mov edi, edi` instruction is replaced and this instruction has been irrelevant to the semantics of the routine, it is, after patching, still possible to use the routine without being redirected: By skipping the first instruction, the routine can still be executed and will yield the same results as before the redirection has been installed.

Setting up the trampoline is not exposed to the issues of runtime code modification as the code being replaced is essentially dead – assuming proper operation, execution should never reach the padding area. Therefore, the only critical runtime code modification is the swapping of the `mov edi, edi` instruction with the short jump.

Not requiring multiple instructions to be patched, this approach is neither exposed to the problems of preemption and interruption (section 4.4.3) nor to basic block boundary-related issues (section 4.4.4).

Moreover, no true disassembly is required by this approach – only the presence of padding area and the `mov edi, edi` instruction may need to be verified. Both can be implemented by a simple memory comparison.

Having entered `Foo` (1) and taken the short jump, execution reaches the trampoline (2). Once the trampoline has been reached, two things have to happen. On the one hand, execution has to be redirected to `CallProxy`, which will initiate the preprocessing of the call. Having five bytes (the size of the padding area) at disposition, using a branch instruction supporting distances large enough to reach this routine directly is feasible.

While the trampoline is specific to the function `Foo`, the routines `CallProxy` and `EntryThunk` are not. In order to share code, all routines that are currently in the state of being traced use these routines. Yet, as call pre- and postprocessing requires knowledge about which routine call is currently being processed, these functions have to be provided the address of the traced routine – which, in this case, is `Foo`.

As an additional complication, it is not clear at this point whether it would be safe to use a register for passing this information or whether doing so would destroy data. Consequently, this information has to be passed on the stack.

As it turns out, given these requirements, placing a near call instruction into the trampoline is a perfect fit. It occupies five bytes, supports a sufficiently large distance and pushes the address of the subsequent instruction onto the stack. This address, in turn, is exactly the address of `Foo`.

Following this call instruction (3), execution reaches `CallProxy`. As its first action, `CallProxy` performs a call to `EntryThunk`. This additional call will become relevant for exit tracing. At this point, the stack contains three return addresses – the actual return address, the address of `Foo` and the return address of `CallProxy`. This is illustrated in figure 6.4.

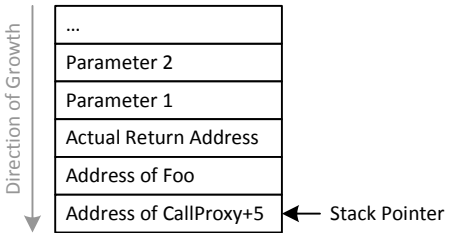

Figure 6.4: Stack contents at entry of CallProxy

All three return addresses serve a specific role for the preprocessing as performed by `EntryThunk`: The actual return address, as noted before, is crucial for being able to return to the caller after the call with all its pre- and postprocessing has been completed. The Address of `Foo` allows `EntryThunk` to distinguish between multiple concurrently traced routines. Finally, the return address of `EntryThunk` itself (`CallProxy+5`) is required for post-processing.

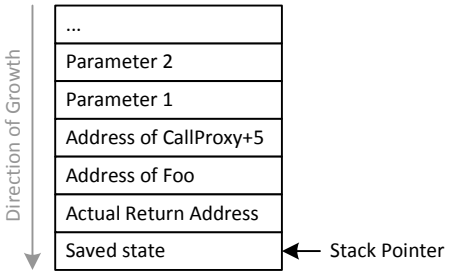

Figure 6.5: Stack contents during operation of EntryThunk

Based on this setup, `EntryThunk` now has the following duties:

- The entry trace event has to be raised, which manifests itself in a callback having to be invoked.
- Execution has to be resumed at the called routine, `Foo` (5). However, doing so would result in an infinite loop as the first instruction of `Foo` is the jump into the trampoline. Therefore, execution has to be resumed at `Foo+2`, the first instruction following this jump.
- Before execution can resume at `Foo+2`, the stack has to be restored – any data that has been pushed onto the stack in addition to the single first return address has to be removed to ensure that `Foo` – in case it makes use of parameters – is able to access these parameters properly.
- The single return address remaining on the stack must not be the actual return address but the address of the code performing post-processing, which is `CallProxy+5`.

The steps taken by `EntryThunk` are therefore as follows:

- The bottom-most of the three return addresses receives the return address of `EntryThunk` itself, which is `CallProxy+5`. Figure 6.5 illustrates the state of the stack after this step has been performed.
- A pointer to the auxiliary stack is obtained. Both the address of the traced routine (`Foo`) and – most importantly – the actual return address is pushed onto this stack.
- The function entry event is raised by invoking an appropriate callback routine.
- The stack is cleaned up. The two pieces of information that are required for post-processing – function address and actual return address – have been safely stored on the auxiliary stack. After trimming the stack, a return to `Foo+2` is performed (5), which results in execution to resume at the traced routine.

`Foo` will eventually return in one of two ways. Either it returns normally or it is aborted prematurely by an exception. Exception handling will be discussed in more detail in section 7.6.

In the usual case, `Foo` will return normally and execution will resume at the return address (6). As a result of the preprocessing, however, the return address points to `CallProxy+5` rather than to the actual calling routine.

Having reached `CallProxy+5`, postprocessing takes place. Two basic tasks have to be accomplished:

- The exit trace event has to be raised, which manifests itself in a callback having to be invoked.
- Execution has to resume at the actual caller.

To perform these tasks, the original return address as well as the address of the traced routine is required. Moreover, such processing requires a certain amount of stack space. Neither of these two addresses is reflected by the stack in its current state. Worse yet, as the calling convention used by `Foo` is unknown, it is not clear either what information can still be expected on the stack and which data the stack pointer points to. This is illustrated by figure 6.6.

Two situations may have occurred:

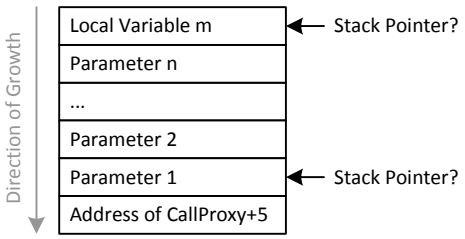

Figure 6.6: Stack contents after return from Foo

- The stack has not been cleaned yet. The callee (i.e. `Foo`) may have used the *C Declare* (*cdecl*) calling convention and thus returned without adjusting the stack pointer to account for the parameters. This situation, for example, occurs when the function returned by executing a `ret`.

  As a consequence, the stack pointer will now point to the first parameter of `Foo`, whose value may or may not still be intact.

- The callee (i.e. `Foo`) has already cleaned the stack. Calling conventions such as *stdcall* require the callee to adjust the stack pointer to account for all parameters. As an example, this situation occurs when `Foo` has returned by executing a `retn m` instruction with m reflecting the number of bytes occupied by its parameters. In this case, the stack pointer will now point to the topmost local variable of the actual callee.

Which of these situations has occurred is not clear. It is, however, possible to deal gracefully with both situations. In both cases, it is safe to assume that the memory above the stack pointer (i.e. lower addresses) may be used as scratch space.

Utilizing this scratch space, a pointer to the auxiliary stack can be acquired and the function address (in this case, the address of `Foo`) as well as the actual return address can be obtained.

Having this information at hand, the exit event can be raised by invoking the appropriate callback routine. After this has been accomplished, the scratch data has to be removed from the stack and register contents have to be restored so that stack and registers are in the same state as they were after return from `Foo`.

Finally, execution has to be transferred to the caller. The crucial point to notice is that regardless of which of the situations listed above has occurred – `CallProxy` is not in charge of cleaning the stack. Therefore, it is safe to push the actual return address and return from `CallProxy` with a simple `ret` instruction. This practice has the additional benefit of not requiring a register or a memory location as an indirect jump would. After the return, execution resumes at the caller as normal.

7 Implementation

7.1 Instrumentation

Instrumenting a routine is a multi-step process. Given the name of a single routine or a pattern matching a set of routines, the first step is to determine the respective virtual addresses. This translation between symbols, i.e. routine names, and virtual addresses requires debugging symbols. As such, it is performed entirely in user mode by the FBT Service DLL, which in turn utilizes dbghelp for symbol handling. To yield valid kernel mode virtual addresses, this process takes the current load addresses of the affected kernel modules into account.

7.1.1 Validation

Once the virtual addresses have been obtained, the FBT Agent is sent an IOCTL command requesting the instrumentation to be performed. It is worthwhile to notice that the content of such IOCTLs – the virtual addresses of routines to be implemented – is delicate from a security standpoint. As these addresses will at least be read from, a malicious client could use such IOCTLs to have the kernel access invalid memory areas, which could in turn provoke a bugcheck.

However, such situations may even occur without malicious intent and access control by itself is therefore insufficient to prevent such situations from occurring: A user may, for instance, accidently request a routine to be instrumented that is part of a driver's initialization code. Such routines, however, are commonly marked with the compiler directive `#pragma alloc_text( INIT, RoutineName )`, which effects that the respective code is discarded as soon as `DriverEntry` has returned. Any attempt to access the code of this routine after driver initialization has completed is likely to provoke a bugcheck.

Thorough validation of instrumentation requests is inevitable to avoid such situations from occurring. Yet, merely validating all memory locations before performing the actual patching operation is insufficient to prevent damage. Memory is a shared resource and other threads may not only modify memory contents concurrently, but also impact the validity of memory at any time – for instance, by unloading a module. Such checks are thus exposed to a *time of check to time of use* (TOCTTOU) [DMS06] issue.

Concurrent unloading of the module to be patched can be assumed to be the most relevant potential source of problems. In user mode windows, such problems could be mitigated by incrementing the *reference count* of the module or *pinning* the module. Unfortunately, no similar API is provided in kernel mode Windows – neither the reference count nor an appropriate lock is accessible through official APIs.

Short of such mechanisms, another approach is used: Some of the checks are delayed until just before the actual patching operation begins. At this stage, execution of concurrent code will be prevented and the IRQL will have been raised to `DISPATCH_LEVEL`. As concurrent unloads are unable to take place under these circumstances, validation can then be safely conducted.

The entire validation logic consists of multiple steps. As a first check, the agent verifies that the addresses received in instrumentation requests indeed fall into memory areas occupied by modules. This check is still performed at `PASSIVE_LEVEL`. While this will detect attempts to access other memory areas such as the pool, it is yet insufficient to deal with addresses pointing to discarded memory or to parts of the module other than a routine. Moreover, the respective module could just be about to be unloaded.

To overcome the limitations of the first step, further validation is performed. However, this validation is, as indicated before, conducted at `DISPATCH_LEVEL` while preventing concurrent execution of other threads. That is, concurrent module unloads as well as pool allocations are prevented from taking place.

First, the validity of memory has to be verified. In the absence of appropriate APIs for this purpose, `MmIsAddressValid` is currently used. Despite its name, this routine checks whether reading or writing a certain address would result in a page fault. In this context, an imminent page fault is an indication for the address being invalid. Yet, in the existence of paged code, a page fault could also be harmless. Short of being able to distinguish between these situations, any address that reading from would incur a page fault is deemed invalid and is rejected – false negatives are accepted in trade for improved robustness.

Having verified the validity of memory, the instrumentability of the routines is verified. That is, the existence of a padding area preceding the function as well as the first instruction being a `mov edi, edi` is checked. Not before these checks have all been passed does the actual patching process begin.

7.1.2 Applying the Patches

Runtime code patching, as discussed in section 4, is exposed to a variety of potential problems. While some of the issues, such as jump distance-related problems or safety concerns related to basic block boundaries and disassembly have already been discussed or even cannot occur with the specific tracing approach chosen, other issues still need to be addressed.

The first issue to be discussed is memory protection. By default, The NT kernel write-protects images to prevent them from being modified during runtime. This write protection could temporarily be revoked by adapting the corresponding page table entries. However, the NT kernel does not provide a public API for performing such modifications. Although accessing the page tables directly is possible, such practice would undermine the role of the *PFN Database Lock*, which in turn makes this approach risky in practice.

A different approach to attain write access to write-protected memory regions relies on *Memory Descriptor Lists* (MDLs). This approach is, as it turns out, also used internally by the hotpatching infrastructure and has been published in [HB05]. The basic idea of this approach is to *lock* the respective memory region to make it resident in physical memory. Then, an additional virtual address mapping for the physical memory backing the respective area is created. If this second mapping permits write access, it is possible to circumvent the protection enforced by the original mapping. Using the second mapping rather than the original mapping, write access is gained while relying on documented APIs only.

The actual patching operation, i.e. conducting the runtime code modification, depicts cross modifying code. As such, the potential issues discussed in section 4.4.1 apply and have to be dealt with.

Using a simple memory move to overwrite this instruction may be considered. Yet, this approach is not viable as there are no guarantees about the location of the respective instruction with respect to memory alignment. Moreover, even when bus locking (i.e. usage of the LOCK prefix) is used to overcome this problem, this practice would not be safe: As discussed in section 4.4.1, such practice could still lead to undefined processor behavior.

The route chosen by NTrace therefore is to idle all processors but the one performing the code patch. That is, by the use of `KeGenericCallDpc`, DPCs are scheduled on all processors. Using appropriate synchronization, the execution of these DPCs is coordinated in a way that allows one processor to conduct the runtime code modifications while all other

processors wait for the former to finish. This strategy reflects the idea of the algorithm defined by Intel for implementing safe cross-modifying code [Int07c].

`KeGenericCallDpc` is similar to `KeInsertQueueDpc`, yet it schedules DPCs on all processors simultaneously and additionally supplies the DPC routine with appropriate synchronization primitives to synchronize their concurrent execution. `KeGenericCallDpc` is undocumented, yet exported by the kernel. As Hoglund and Butler have shown, however, a similar mechanism can be also implemented based on regular DPCs [HB05].

Once the DPC having been chosen to conduct the runtime code modification begins executing, the mentioned final step of validation can be performed. As discussed before, it is crucial for this step to be taken while all other concurrent activity on this system has been paused.

However, at this point, it should become clear that the validation process is still exposed to an – albeit less severe – TOCTTOU issue: Locking and mapping the memory using a MDL has to be performed below `DISPATCH_LEVEL` to account for the possibility of paged code. It is therefore not possible to perform the locking from within the DPC. `MmProbeAndLockPages`, unfortunately, is unable to cope with all types of invalid addresses. Therefore, the addresses have to be checked using `MmIsAddressValid` first. Yet, due to concurrent activity, the result of the validity check may already be outdated when `MmProbeAndLockPages` is called. Given the restrictions of the API, however, this race condition seems hardly avoidable.

Having completed validation, each DPC raises the IRQL above all device IRQLs in order to protect against interrupts. Performing the actual patch then merely is a matter of copying memory using `memcpy` – the fact that `memcpy` copies non-atomically is not of concern.

When all patches have been applied, the IRQL can be lowered again, all other DPCs can be unblocked and execution can resume on all processors. Yet, before the DPCs return, a final step is required in order to comply with the requirements for cross modifying code [Int07c]: To flush all modifications, a serializing instruction, namely `cpuid`, is called by each DPC routine.

To amortize the cost of the entire patching process, multiple routines can be instrumented at once.

7.2 Auxiliary Stack

A cornerstone of NTrace is the auxiliary stack. Whenever an instrumented routine is entered, a stack frame will be pushed onto this stack. During postprocessing of the routine, the frame will be popped and the information contained is used to continue execution.

Two basic pieces of information are stored in a stack frame: The address of the routine being executed and the original return address. As discussed in section 6.2.2, the stack-located return address of the traced routine is replaced as part of the call preprocessing. Yet, in order to properly continue execution after return from the routine, the return address has to be restored during call postprocessing. For this purpose, the address is preserved as part of this stack frame.

The rationale of storing the routine address is less obvious. For keeping execution flow intact, this address is not required. Rather, its sole purpose is to be included in the event record written during call postprocessing. As will be discussed in more detail in section 7.7.4, this simplifies the consumption of events and also allows a certain degree of compensation for lost events.

The auxiliary stack has a fixed size of 256 frames. To avoid overflows from occurring, the entry thunk, before allocating a stack frame, checks for this condition. In case of stack

depletion, it fails gracefully by falling back to not tracing the call: The return address and stack modifications are reverted, so that no call postprocessing will have to take place and thus, no stack frame is required. Execution is then redirected to the actual routine, which will execute without being traced.

7.3 Thread Data

Containing information specific to a single thread, the auxiliary stack needs to be maintained on a thread-local basis. However, the auxiliary stack is not the only piece of information that needs to be kept track off for each thread.

For this reason, all thread-local state is grouped together in a single data structure, the *Thread Data*. Each thread on which tracing activity occurs is attached a dedicated instance of this structure.

7.3.1 ETHREAD Association

User mode Windows allows *Thread Local Storage* (TLS) to be used for maintaining thread-local information. However, being a user mode concept, no similar facility exists for kernel mode Windows.

TLS is maintained as part of the *Process Environment Block* (PEB) and *Thread Environment Block* (TEB). Private to each thread, the TEB contains a plethora of thread-specific information and reserves space for use by TLS slots, which are defined in the PEB. PEB and TEB are maintained by the kernel, yet they are mapped into user mode address space and can thus be accessed from either mode.

Although both PEB and TEB can be managed from kernel mode and utilizing TLS from kernel mode seems feasible, it is worth pointing why this approach is not viable.

The first obstacle lies in the fact that TLS slots are maintained on a per process basis. Yet, to allow TLS data to be available in an arbitrary process context, a TLS slot would have to be allocated for each process. To account for the fact that slot numbers are likely to differ among processes, a mapping between process and slot would have to be maintained as well.

But even if this problem was solved, the approach would fail to work for system threads. System threads are used in kernel mode only and therefore lack certain properties a user mode thread features – which includes the TEB. Having to implement a different approach for system threads, however, raises the complexity of this approach and renders it impractical.

Threads are represented by the Windows Kernel as KTHREAD structures. The KTHREAD structure is embedded into a ETHREAD structure, which adds additional information used by the kernel subsystems. As such, KTHREAD/ETHREAD structures are not only used for scheduling but also serve as thread local storage for the kernel and its subsystems.

Drivers, however, are not offered a similar means to maintain thread local state. Not only does the ETHREAD not reserve space for use by drivers, it is also documented as having to be considered opaque. That is, its layout must be expected to be subject to change.

7.3.1.1 Windows Research Kernel

On the Windows Research Kernel, this situation is easily dealt with. Having access to the sources, the definition of ETHREAD has been enhanced by an additional field, *PVOID ReservedForFbt*. This field is used to store a pointer to the Thread Data structure.

7.3.1.2 Retail Kernel

Short of being able to modify the ETHREAD structure, a different strategy has to be found for the retail edition of the Windows kernel.

Initially, a hashtable had been chosen for implementing a mapping between ETHREADs and Thread Data structures. Although workable, this approach had a number of disadvantages: As lookups can be expected to be performed at a very high frequency – at least once per entry or exit of a traced function – the nature of a hashtable of offering non-constant lookup time is unfortunate. Moreover, access to the hashtable has to be interlocked properly. Not only must this locking be workable at any IRQL, which rules out all dispatcher-based locks, it also has to properly guard against concurrent access by interrupts. As a consequence, regardless of the exact locking scheme used, accessing the table has to involve raising the IRQL. Yet, as adjusting the IRQL is a non-trivial operation, the frequent calling of `KfRaiseIrql` and `KfLowerIrql` must be expected to have a negative performance impact.

Another non-obvious drawback of the callout to `KfRaiseIrql` and `KfLowerIrql` is potential interference with Driver Verifier. If Driver Verifier has been enabled for the agent driver, `KfRaiseIrql` is among the routines hooked. Unfortunately, the surrogate implementation, `VerifierKfRaiseIrql`, itself calls a number of memory-related routines, `MiTrimAllSystemPagableMemory` being one of them. If one of these routines has previously been instrumented for tracing, this leads to endless recursion. This situation is illustrated by the stack trace in figure 7.1, which was taken shortly before a stack overflow occurred.

```
1  [...]
2  nt!ViTrimAllSystemPagableMemory+0x53
3  nt!VerifierKfRaiseIrql+0x39
4  jpkfar32!JpfbtsAcquireTlsLock+0xd
5  jpkfar32!JpfbtGetFbtDataThread+0x11
6  jpkfar32!JpfbtpGetCurrentThreadDataIfAvailable+0x11
7  jpkfar32!JpfbtpGetCurrentThreadData+0x11
8  jpkfar32!JpfbtpGetCurrentThunkStack+0xb
9  jpkfar32!JpfbtpFunctionEntryThunk+0x17
10    (instrumented nt!MiTrimAllSystemPagableMemory)
11 nt!ViTrimAllSystemPagableMemory+0x53
12 nt!VerifierKfRaiseIrql+0x39
13 jpkfar32!JpfbtsAcquireTlsLock+0xd
14 jpkfar32!JpfbtGetFbtDataThread+0x11
15 jpkfar32!JpfbtpGetCurrentThreadDataIfAvailable+0x11
16 jpkfar32!JpfbtpGetCurrentThreadData+0x11
17 jpkfar32!JpfbtpGetCurrentThunkStack+0xb
18 jpkfar32!JpfbtpFunctionEntryThunk+0x17
19    (instrumented nt!MiTrimAllSystemPagableMemory)
20 nt!ViTrimAllSystemPagableMemory+0x53
21 nt!VerifierKfRaiseIrql+0x39
22 jpkfar32!JpfbtsAcquireTlsLock+0xd
23 [...]
```

Listing 7.1: Stack trace illustrating recursion caused by interference of Driver Verifier with a hashtable based approach

In favor of a more lightweight solution that is also less prone to reentrance, a different approach had to be taken. Ideally, the pointer to the Thread Data structure could be stored in the ETHREAD, as in the case of the WRK. Yet, ETHREAD does not reserve space for this and is also a rather dense-packed structure, i.e. no spare space for storing a full pointer-sized value could be located.

At least two fields of the ETHREAD, `SameThreadPassiveFlags` and `SameThreadApcFlags`, however, are wider than necessary. Both fields are defined as ULONG, yet less than 16 bits

are actually used. The pointer to the Thread Data structure is thus split into two 16 bit words and each word is stored in the upper word of the two respective fields. The contents of the lower words are preserved.

Both fields are explicitly documented in the WRK sources as being accessed from the same thread only. That is, modifying the contents of these fields can be expected to be safe without having to guard against concurrent access. To protected against reentrance, however, it is still crucial to perform each of the two stores in an interlocked fashion.

To account for the fact that the layout of ETHREAD evolves over operating releases, the offsets to the fields are chosen based on the individual build of the kernel.

Using spare bits in the ETHREAD structure has proven to be a workable and robust approach, yet it clearly does not quite follow good practice. However, as has been demonstrated in case of the WRK, a similar, yet cleaner approach exists. Therefore, having to revert to using spare bits does not denote an inherent limitation of the entire tracing approach but is rather an artifact of the inability to augment the source code of the retail kernel.

7.3.2 Allocation

Being able to dynamically load and initialize the tracing system on a running system was among the aims of the implementation. When the agent driver is first loaded, all threads can be expected to not have a Thread Data structure attached. Yet, before the first procedure entry event can be captured, the respective thread must have been attached such a structure.

One option to implement timely allocation and association of Thread Data structures would be to eagerly attach each existing thread on the system such a structure during initialization of the driver. Each thread being spawned after the driver has been loaded would also have to be attached a Thread Data structure. Despite the various race conditions this approach would have to cope with, it also is inefficient: If only a few threads are affected by the instrumentation, most of the allocated structures will not be used.

Lazy allocation of Thread Data does not suffer from these problems and has thus been favored. Not before the first procedure entry has been intercepted, is the structure allocated for the current thread.

However, lazy allocation introduces a different set of problems. For one, the Thread Data has to be allocated from the non paged pool. `ExAllocatePoolWithTag`, the routine to allocate such memory, allows allocations to be made at IRQL `DISPATCH_LEVEL` and below. It is, however, well possible that the first routine to be traced on a thread runs at a higher IRQL – which, for instance, is the case for an interrupt service routine. As this is a rather common scenario, failing the allocation under such circumstances is undesirable. To remedy this problem, a pool of preallocated structures is created during driver initialization, which is used to satisfy allocation requests at elevated IRQL.

While this approach works decently when only drivers are instrumented and traced, it becomes risky as soon as routines of the kernel image are instrumented.

`ExAllocatePoolWithTag` is a non-trivial operation and utilizes a number of other routines. If `ExAllocatePoolWithTag` itself or any of these routines is instrumented, happens to be the first instrumented routine hit by a specific thread, and the IRQL is below `DISPATCH_LEVEL`, it will cause an allocation to be made. Yet, calling `ExAllocatePoolWithTag` to perform this allocation will cause the same procedure to be traced again. As there is still no Thread Data available, an allocation will be triggered. At this stage, a point of endless recursion has been reached, which will finally lead to a stack overflow and a bugcheck.

The initial approach to avoid allocation-related issues was to protect against reentrant Thread Data allocations. Before `ExAllocatePoolWithTag` is called, the current thread is flagged as being in the state of allocating a Thread Data structure. Whenever such an allocation is attempted, this flag is checked. In case it is set, reentrance must have occurred and the allocation request is failed before any harm can be done. Although some events will be lost, the danger of endless recursion is mitigated. After a Thread Data allocation has been completed, the flag is cleared.

It is, however, worth pointing out that although being an effective countermeasure to reentrance, calling `ExAllocatePoolWithTag` in presence of instrumented kernel routines may still endanger system stability: A situation that has been encountered in practice was that one of the routines indirectly used by the background threads of the memory manager was instrumented and happened to be the first instrumented routine to be hit on this specific thread. To allocate a Thread Data structure, `ExAllocatePoolWithTag` was called. However, the special circumstances of this call being made from one of the memory manager background threads in the midst of their operation led to a situation where `ExAllocatePoolWithTag` was confronted with an invalid state, which manifested itself in a `IRQL_NOT_LESS_OR_EQUAL` bugcheck.

Taking consequences of this failure, performing lazy allocations using `ExAllocatePoolWithTag` must be expected to be too risky in this context: While the approach can still be used in case only drivers are traced, it should be avoided as soon as kernel routines are instrumented. Hence, the FBT core library allows a flag to be set that causes all allocations, regardless of the IRQL, to be satisfied using the preallocation pool.

7.3.3 Deallocation

Once the agent driver is requested to unload, one of its duties is to clean up all resources previously allocated. Keeping track of all Thread Data structures that have been lazily allocated before is thus crucial in order to allow an orderly cleanup.

Before a Thread Data structure is attached to a thread, it is put onto a list: This list, rooted in the global structure where the agent maintains its state, contains all allocated Thread Data structures currently in use.

When the driver is unloading and it has been verified that deallocating the structures is indeed safe, this list of Thread Data structures is traversed. First, each structure is disassociated from the corresponding thread – this is crucial to avoid leaving a stray pointer in the thread's ETHREAD, which would lead to problems when the agent is loaded again at a later point in time. Depending on the pool from which it has been allocated (preallocation pool or normal pool allocation), the structure is then freed.

Another aspect of deallocation is dealing with threads terminating before the driver is unloaded. If such occurrences were ignored, the deallocation logic depicted above could touch freed memory – memory that has previously been occupied by an ETHREAD but has meanwhile been freed.

For this reason, the agent additionally registers for notification about terminating threads by using `PsSetCreateThreadNotifyRoutine`. Once notified, the agent will cleanup and unregister the Thread Data structure of the respective thread.

7.4 Unloading

Before the Function Boundary Tracing Agent Driver can be unloaded, all outstanding instrumentation has to be revoked. For this to work, the FBT core library maintains a list

of all active code patches, the *patch database*. Before unloading, all of these patches are revoked by swapping the old code, which is also preserved as part of this database, back in. The actual code modification in conduced in the same manner as has been discussed in the context of instrumentation.

Yet, such operation could be risky in case the affected routine is part of a driver rather than part of the kernel itself. In this case, it is possible that the driver has meanwhile been unloaded, which means that the affected memory is not valid any more. To avoid invalid memory accesses, the validity of memory is checked once again before the code replacement is conducted.

A related, yet less common situation is where the driver has been unloaded, but the affected memory region has meanwhile been reused for different purposes. To detect this situation, the memory is additionally checked for containing the characteristic jump-to-trampoline instruction.

However, merely reverting all routines to their original state is insufficient to ensure that unloading the driver is safe. The following example illustrates this problem:

- Thread 1 is currently executing Routine `Foo`, which has been instrumented for tracing.
- Thread 2 revokes instrumentation on `Foo` and unloads the driver.
- Thread 1 reaches the end of `Foo` and will return into the `CallProxy`. As `CallProxy` is part of the driver, which has meanwhile been unloaded, this will result in an access violation or arbitrary memory contents to be executed.

The driver, being dynamically loadable and unloadable, can be considered dynamically allocated code. The driver code being executed as part of pre- and postprocessing of every traced routine is thus fully exposed to the issues of lifecycle management discussed in section 4.4.6. As such, the situation illustrated merely reflects the inherent race condition already described.

As indicated by section 4.4.6, the best the driver can do in such situations is to try to minimize the risk by tracking usage of the dynamically allocated code. Although such practice cannot remedy the problem in its entirety, delaying the unload until the usage count has dropped to zero decreases the likelihood of such problems significantly.

As a fortunate side effect of the tracing mechanism implemented and discussed so far, introducing additional reference counting to implement such usage tracking is not necessary. Rather, the auxiliary stacks of all threads can be checked for being empty. If any of the stacks is non-empty, execution is delayed by one second and the check is retried.

Like reference counting, this mechanism is effective, yet still exposed to race conditions. On the one hand, the implicit assumption of that the auxiliary stacks cease to grow as soon as all routines have been uninstrumented is flawed: It is possible that when uninstrumentation occurs, some thread may already have entered the `EntryThunk`, but is preempted before being able to allocate a frame on the auxiliary stack. Resuming execution at a later point in time, a stack frame allocation will be conducted although all instrumentation has meanwhile been revoked.

On the other hand, it is possible that the last frame on the auxiliary stack has been removed, yet execution has not yet left the CallProxy. Again, the driver code is still in use although all auxiliary stacks may already be empty.

Unable to avoid these race conditions, the driver follows the approach as taken by KernInst [TM99, Tam01] and introduces a further three-second delay. During these three seconds, all threads are given the chance to leave the critical code sections.

Although still not entirely safe, this unloading strategy has worked well in practice. However, especially when instrumenting large amounts of routines, it turns out that it may in fact take a significant amount of time before all traced routines have completed. It is thus not unusual for the driver unload to take several seconds or minutes.

When instrumenting the kernel itself, this becomes even more noticeable, as the kernel contains a number of routines that may block for significant amounts of time. One such routine is `KeWaitForSingleObject`. When used for a wait that takes an hour to be satisfied – possibly on behalf of a user mode application – it will not return before this hour has elapsed. If `KeWaitForSingleObject` has been instrumented and the driver is later requested to be unloaded, unloading will also be delayed until this wait has been satisfied.

Not before these safety checks have been passed, the driver begins to tear down. This includes disassociating ThreadData structures from their threads and freeing the respective allocations, as well as flushing buffers.

7.5 CallProxy and CallThunk

The general working of the CallProxy and CallThunk routines and the way they are used to intercept function invocation has been discussed before in section 6.2.

However, a number of details about the implementation of these routines have not been discussed so far.

7.5.1 State Preservation

The aim of the CallProxy and the EntryThunk is to interpose the call to a routine. Yet, in order not to change the outcome of the call, it is crucial that the *contract* between the calling and the called function remains intact.

In this regard, the *contract* between the two function is defined by the calling convention. The calling convention defines in which memory locations and in which order parameters have to be passed, where the return value of the function is to be stored and whose duty it is to clean up stack space that might have been occupied by parameters.

Regarding the WRK sources and the header files of the WDK, the most prominent calling convention encountered in the kernel is *stdcall*, followed by *fastcall*, which is used rather infrequently.

One approach to ensure that the contract between two functions remains intact could be to rely on the call adhering to a certain calling convention. Having access to public symbols only, it is not clear which calling convention is used. Yet, it is reasonable to assume that it is either *stdcall*, *fastcall*, *cdecl* or *thiscall*. Based on this assumption, it would, for instance, be safe to not preserve the contents of the `eax` register during call preprocessing. `eax` is volatile and none of the mentioned conventions use it to pass parameters. In the same manner, `ecx` would not need to be preserved during postprocessing. Again, `ecx` is volatile and is not used to store the return value by any of these calling conventions.

Although appealing, this approach is not feasible in practice. For *exported* routines, i.e. routines made available for use by a different module, the compiler indeed makes sure that the calling convention is adhered to. For routines used internally, however, the compiler is free to ignore the specification of a calling convention: Rather than passing all parameters

on the stack as the respective calling convention may specify, the compiler may, for instance, decide to pass one or more parameters in registers.

It is worth pointing out that not only `static` routines, i.e. routines used within a single object file only are subject to such optimization. With the advent of link time code generation, the Microsoft compilers are capable of performing such interprocedural optimization on the entire image, rather than on individual objects only. As a consequence, any routine not exported by the respective module must be assumed to potentially not adhere to any of the default calling conventions.

Aiming at being able to trace internal routines as well as exported routines, this approach has thus been dismissed. Yet, for a solution that restricts itself to intercepting exported routines such as *IAT hooking*, this approach may in fact be viable.

Short of any guarantees by the compiler, a rather pessimistic approach had to be taken:

- Not knowing which registers are used to pass parameters, which registers are non-volatile and which registers are free, the assumption is that all registers contents have to be preserved – both during call pre- and postprocessing.
- It is assumed that an arbitrary amount of stack space is used for passing parameters. Whether the caller or the callee is in charge of cleaning the stack is unknown.
- All memory beyond the current stack pointer (i.e. higher addresses) is assumed to be unused.

The last item requires further discussion. The stack pointer points to the topmost item on the stack. Convention dictates that stack memory beyond the location pointed to by the stack pointer, should not be accessed. Still, such memory locations can be easily addressed and the compiler could generate code that does so.

There is, however, a simple reason why it is safe to assume that kernel mode code does not defy this convention: Interrupt processing.

If an interrupt occurs and no privilege-level change has to be performed, the CPU will push the EFLAGS, CS and EIP registers on the stack [Int07a]. That is, the stack of whatever kernel thread happens to be currently running on this CPU is reused and the memory locations beyond the stack pointer will be overwritten by these three values. Handling the interrupt, which involves running the *interrupt service routine* (ISR) occurs on the same stack as well.

As a consequence, code accessing memory locations beyond the current stack pointer would have such memory locations be overwritten in case of interruption. As the compiler can be assumed to be unaware of which code sequences are subject to interruption and which code sequences are not, it is safe to assume that the compiler will always emit code that adheres to this convention. The same holds for hand-written assembly code sequences, although these routines tend to be not hotpatchable anyway.

The consequence of this observation is that the CallProxy and EntryThunk can safely use the stack as scratch space as long as they adjust ESP accordingly. Due to the restriction of having to preserve all register contents, this ability to use the stack is essential. It allows both the CallProxy and EntryThunk to establish a stack frame and to free a certain amount of registers by pushing their contents onto the stack and restoring them later.

7.5.2 Reentrance

During call pre- and postprocessing, the state of the thread-local ThreadData structure is modified. This modification, which includes, but is not limited to setting up a frame in the auxiliary stack, is a multi-step process. Although being consistent at the beginning and at the end of this process, the ThreadData structure will temporarily be inconsistent during this modification process.

This temporary inconsistence will become an issue as soon as the thread is interrupted while being in the midst of a ThreadData modification process. If any routine that is being executed as part of the interrupt processing has been instrumented for tracing, reentrance will occur. That is, the preprocessing of the call to such a routine will operate on an inconsistent ThreadData structure, which leads to arbitrary behavior.

However, not only the modifications to the ThreadData structure themselves are sensitive to reentrance. Lazy allocation of these structures is equally affected. Even when the allocation code is fully reentrant, situations can occur where two structures are allocated for the same thread. Management of buffers, a topic discussed in section 7.7 is affected in a similar manner.

Properly handling reentrance is thus crucial. The two options are thus to either make all affected code fully reentrant or to properly guard against reentrance. While the first option may in fact be feasible, the affected routines of the current implementation only partially achieve this aim. That is, both the CallProxy and the EntryThunk contain a window of instructions that must be protected from reentrance.

To attain this protection, a simple scheme has been implemented. Leveraging one of the bits borrowed from the ETHREAD, a flag has been introduced. Before entering the critical window, the flag is attempted to be set; it is cleared as soon as the window is left. If the flag turns out to be already set when entering the window, reentrance must have occurred. Similar to what occurs on auxiliary stack overflow, any modifications are reverted and the respective routine is executed without being traced.

It is worth pointing out that the situations depicted above do in fact occur on a regular basis. In practice, before appropriate countermeasures had been put in place, and all kernel routines (i.e. `nt!*`) were instrumented, it was only a matter of seconds before these problems led to a triple fault.

7.6 Exception Handling

Windows NT features a builtin exception handling infrastructure, *Structured Exception Handling* (SEH). SEH exceptions are not only used for handling *software exceptions*, i.e. exception explicitly raised via `RtlRaiseException`. It is also used for handling hardware-defined faults or traps such as *Integer divide by zero* conditions.

So far, the existence of exceptions has largely been ignored in the discussion of function entry and exit tracing. Rather, the implicit assumption has been that for each function entry successfully intercepted, there will be an intercepted function exit. During preprocessing, a frame is pushed onto the auxiliary stack, during postprocessing, a frame is popped. As entry and exit interceptions cleanly nest, it is assured that the popped frame is in fact the right one, i.e. the frame corresponding to the specific call frame.

Exceptions, however, thwart this assumption. When a routine raises an exception, the exception can cause the routine to not run to completion but to be left prematurely. The same holds for all routines corresponding to the call frames between the call frame handling the exception and the call frame raising the exception. All these routines will be left

prematurely as well. Prematurely leaving a routine means that it does not return to the caller as normal. This, in turn, becomes a problem if one of the routine has been instrumented for tracing: a skipped return implies that postprocessing will not be able to take place and thus, that the corresponding frame will not be popped from the auxiliary stack.

An unbalanced auxiliary stack entails three basic issues. The least of which is a resource leak: The un-popped frames will remain on the stack and new frames will be pushed atop of these frames. Preventing the auxiliary stack from ever becoming empty again, these frames effectively denote a resource leak. Diminishing the effective size of the stack, they also raise the probability of stack depletion. As discussed in section 7.2, such auxiliary stack depletions can be properly dealt with. However, they will still lead to more events being missed.

The second issue is an implication of the frames being leaked. As the auxiliary stack will never empty again, the unloading logic discussed in section 7.4 will keep identifying an affected thread as still being active: Short of being able to distinguish leaked frames from non-leaked frames, it will assume that the thread is still running traced routines. Waiting for these routines to complete, however, will be unfruitful – the unload will take forever.

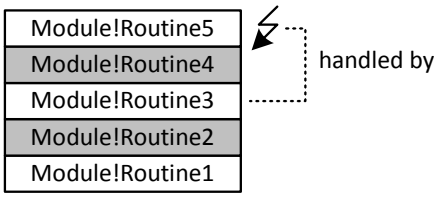

Figure 7.1: An example callstack when an exception is raised. Gray boxes denote stack frames of traced routines.

The most critical issue, however, is what occurs in situations where there are stack frames of traced routines on the callstack both, between the stack frame of routine raising and the stack frame of routine handling the exception, as well as underneath the stack frame of the routine handling the exception. An example for such a situation is illustrated by figure 7.1: Depicting a callstack, the stack is drawn growing bottom-up. Each box denotes a call frame, gray boxes denote call frames of traced routines. `Routine5` raises an exception, which is handled by `Routine3`. As discussed before, one frame will now be leaked on the auxiliary stack. If `Routine3` and `Routine2` later return, postprocessing for the latter routine has to take place. That is, the topmost frame is popped from the auxiliary stack and is used to obtain and reconstruct the original return address. However, as the top frame denotes the leaked frame and is thus not the frame corresponding to the call frame of `Routine2`, this address will be wrong – rather than pointing into `Routine1`, it will point into `Routine3`. Needless to say, continuing execution at the wrong return address will lead to behavior that may be considered arbitrary.

Properly dealing with exceptions to avoid such interferences is thus of utmost importance. The implementation has to ensure that the auxiliary stack is always kept in balance and in sync with the actual call stack.

7.6.1 Structured Exception Handling

SEH is available both in kernel and user mode. While user mode SEH is not within the scope of this discussion, two things are worth pointing out: First, handling user mode

exceptions requires the help of the kernel. That is, user mode exception handling is not a pure user mode concept. Second, both, user and kernel mode exception dispatching, are sufficiently similar to be discussed jointly.

The workings of user mode SEH have been discussed in detail by [Pie97]. Given their similarity, this discussion largely holds for kernel mode SEH as well.

To allow a more detailed discussion of SEH and how it is relevant to function boundary tracing, the following section provides a brief overview of the inner workings of SEH. It is, however, crucial to notice that the implementation of SEH on IA-32 differs fundamentally from the implementation on AMD64. The remaining discussion is therefore largely IA-32-specific.

SEH is synchronous in that throwing and handling an exception only affects a single thread. It is also built upon the notion of frame based exception handlers. That is, exceptions are not handled globally – rather, each routine may associate an individual exception handler with its call frame.

Short of hardware support on IA-32, SEH uses a programmatic approach to maintain exception handlers and their association to call frames. Each routine wishing to install an exception handler – either to actually handle an exception or at least to be notified about an exception, does so by creating an `EXCEPTION_REGISTRATION_RECORD`. This structure holds a function pointer to the exception handler routine that is to be called during exception dispatching. To become effective, the structure is registered by placing it in thread-locally maintained linked list of registration records. These steps are usually done at the very beginning of a routine. Accordingly, before the routine is left, the record must be unregistered by removing it from the linked list. The structure itself *must* be allocated on the stack.

The root of the list, i.e. the pointer to the topmost registration record is located at the very beginning of the *Thread Information Block* (TIB). In user mode, the TIB is part of the TEB, in kernel mode, it is part of the *Processor Control Region* (PCR)[1]. In both cases, it is always accessible at offset 0 in the FS segment. Being part of the PCR rather than the KTHREAD, the dispatcher ensures that the pointer is updated whenever a different thread becomes subject to execution.

7.6.2 Exception Dispatching Process

When an exception is raised, the system walks the list of registration records, calling each handler routine until it finds a handler agreeing to handle the exception. This is implemented in `RtlDispatchException`. Focusing the discussion on the usual cases only, a handler routine will return either `ExceptionContinueSearch`, indicating that the search for a handler shall continue, or `ExceptionContinueExecution`, indicating that the handler has handled the exception. Certainly, before a handler can return `ExceptionContinueExecution`, a number of steps must have taken place.

There are basically two ways how an exception handler can deal with an exception. First, the handler can fix the reason for the fault and can request the faulting instruction to be retried by retaining the instruction pointer and returning `ExceptionContinueExecution`. Regarding traced routines, this situation can be deemed harmless.

The second, more common case is that execution is to be resumed elsewhere – such as in some exception compensation code located in the routine having installed the respective handler. However, continuing execution at a different address not only requires the instruction

[1] The PCR maintains processor-specific state. The kernel maintains one instance of this structure per processor.

pointer to be updated and returning `ExceptionContinueExecution`, it also implies that the routines corresponding to any outstanding call frames are about to be left prematurely. This is the situation where frames on the auxiliary stack are in risk of being leaked.

NT provides a dedicated routine for this purpose, `RtlUnwind`. Before internally calling `ZwContinue` to perform the continuation, `RtlUnwind` performs a second phase of exception dispatching, *unwinding*. During unwinding, each routine about to be prematurely left is given the option to perform cleanup work. That is, the list of exception registration records corresponding to the call frames in question is walked once more, and each handler is called again with a flag indicating that unwinding is taking place. Not before this phase has been completed is execution resumed at the new location.

Based on this brief summary of the exception dispatching process, it becomes clear that whenever unwinding occurs, the auxiliary stack has to be unwound as well.

7.6.3 Auxiliary Stack Unwinding

Adapting `RtlUnwind` in the WRK to additionally trigger unwinding of the auxiliary stack seems possible, yet would have thwarted the intent of implementing the entire solution as a device driver and making it compatible with the retail kernel. This approach has hence been dismissed.

To be notified about a call frame corresponding to a traced routine being unwound, the natural approach is to leverage the SEH unwinding infrastructure itself. The basic idea is thus as follows: During preprocessing of a traced routine, an additional exception handler is registered. Following general SEH practice, the handler will be unregistered during postprocessing.

While this handler will merely return `ExceptionContinueSearch` for all exceptions being dispatched, it will take additional action in case of an unwind: To keep the auxiliary stack up to date, all it has to do is to pop its topmost frame.

Unfortunately, this approach is not viable in practice. Registering an additional exception handler requires a `EXCEPTION_REGISTRATION_RECORD` structure to be set up. As noted before, SEH requires this structure to be allocated on the stack and be accessible during the duration of the entire call. Yet, although the structure occupies only 8 bytes, such stack allocation, as has been discussed in section 6.2, is infeasible to be made.

Any attempts to store the registration record elsewhere, such as as part of the current frame in the auxiliary stack, are thwarted by the thorough validation logic implemented both in `RtlDispatchException` and `RtlUnwind`.

Although the original idea kernel has been retained, the implementation therefore had to be adapted in order to account for these special circumstances. Short of being able to allocate a dedicated registration record, the next underlying registration record is *hijacked*: The existing pointer to the exception handler is exchanged against a pointer to the *auxiliary stack unwinding exception handler*. This exchange is conducted during preprocessing of the traced routine. Accordingly, the modification is reverted during postprocessing.

The handler itself, after completing its own work, will delegate the call to the original handler having been replaced. That way, both handlers effectively share the same registration record.

Certainly, this scheme requires the pointer to the original exception handler to be retained. Yet, the actual registration record does not offer space for storing an additional pointer, so it has to be stored separately. Conceptually, the right kind of storage would be a thread-local map between pointers to registration records and pointers to the corresponding original handler routines.

In practice, however, a full-fledged map can be assumed to be not necessary. The routine being looked up most frequently will always be the routine corresponding to the topmost stack frame, followed by the routine of the second most recently pushed stack frame, and so on. As a consequence, the implementation does not use a separate map, but rather stores this information as part of the current stack frame of the auxiliary stack.

To account for cases where the topmost frame does not contain the pointer being looked for, the stack frame has been augmented by a pointer to the corresponding exception registration record. That way, looking up the correct handler is a matter of scanning the stack top-down, checking each pointer to the registration record, and finally obtaining the corresponding pointer to the handler routine. As indicated before, this scan can be expected to span few frames only, in most cases only the topmost frame.

Listing 7.2 shows the abbreviated structure defining an auxiliary stack frame.

```
1  typedef struct _JPFBT_THUNK_STACK_FRAME
2  {
3    //
4    // Hooked procedure.
5    //
6    ULONG_PTR Procedure;
7
8    //
9    // Caller continuation address.
10   //
11   ULONG_PTR ReturnAddress;
12
13   struct
14   {
15     //
16     // Pointer to the (stack-located) registration record
17     // corresponding to this stack frame.
18     //
19     PEXCEPTION_REGISTRATION_RECORD RegistrationRecord;
20
21     //
22     // Pointer to original handler that has been replaced.
23     //
24     PEXCEPTION_ROUTINE OriginalHandler;
25   } Seh;
26 } JPFBT_THUNK_STACK_FRAME, *PJPFBT_THUNK_STACK_FRAME;
```

Listing 7.2: Definition of an auxiliary stack frame

So far, we have implied that using a single exception registration record for both the original and the auxiliary stack unwinding exception handler, and performing proper call delegation yields the same functionality as using two separate registration records. However, this is not the case – in fact, there is a slight semantic difference that requires the scheme to be revised.

7.6.3.1 Topmost Exception Handler Initiating an Unwind

Figure 7.2 illustrates a situation in case no tracing is used. One of the call frames (Routine3) has set up a SEH record. For the sake of simplicity, this is the only record on the list of exception registration records pointed to by the processor's PCR.

The same situation shall now be considered under the assumption that Routine4 has been instrumented for tracing. Following the technique discussed before, the SEH record of Routine3 has been updated to point to the auxiliary stack unwinding exception handler.

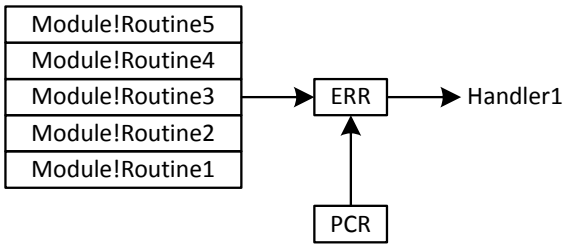

Figure 7.2: An example callstack. ERR: Exception Registration Record.

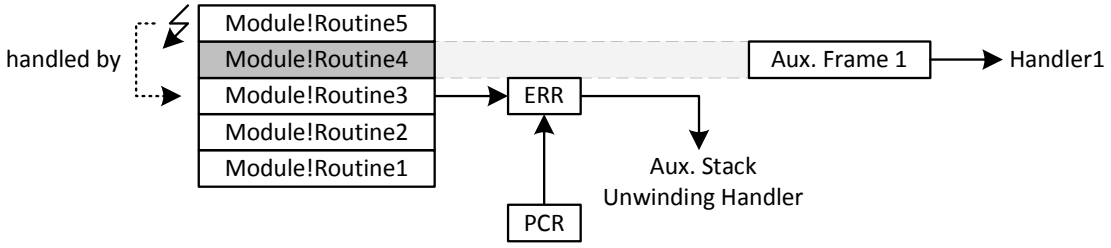

Figure 7.3: An example callstack containing a call frame of an instrumented Routine, Routine4.

The pointer to the original exception handler has been stored in the auxiliary stack frame corresponding to Routine4. The resulting setup is illustrated in figure 7.3.

If Routine5 now raises its exception, `RtlDispatchException` will first inspect the SEH Record of Routine3 and invoke its handler, which is the auxiliary stack unwinding exception handler. Not yet knowing whether the exception will trigger an unwind or not, this routine will merely delegate the call to the original handler, Handler1, in order to let it make a decision.

We now assume that the original handler decides to handle the exception by continuing execution in Routine3 and calls `RtlUnwind` accordingly. However, `RtlUnwind`, noticing that it has been the handler of the topmost registration record that has handled the exception, realizes that no unwinding has to take place and will initiate the continuation. While this behavior is clearly correct, the net result is that the unwinding logic of the auxiliary stack unwinding exception handler has not been invoked.

The underlying problem is that for the scheme to work, `RtlUnwind` would have to perform unwinding for all registrations down to *and including* the registration whose handler has handled the exception. Only in this case would it invoke the unwinding logic of the auxiliary stack unwinding exception handler. However, as said, `RtlUnwind` rightfully excludes the latter from unwinding.

Unaware of the fact that the registration record in question is actually shared by two handlers, this leads to the situation where the auxiliary stack unwind fails to take place.

As such, the mechanism as discussed so far is not complete. There is, however, a way to mitigate this problem and cause `RtlUnwind` to always properly call the auxiliary stack unwinding exception handler. The idea is as follows: The auxiliary stack unwinding handler

is split into two routines: The *proxy exception handler* and the *unwind handler*, both valid exception handler routines.

The unwind handler merely contains the auxiliary stack unwinding logic. That is, when called, the routine unconditionally pops the topmost frame. Moreover, an appropriate event callback routine is invoked.

The proxy exception handler, as its name suggests, delegates to the original exception handler. When called in the non-unwinding case, however, before the call is delegated to the original routine, an additional, full-fledged SEH frame is set up, specifying the unwind handler as exception handler.

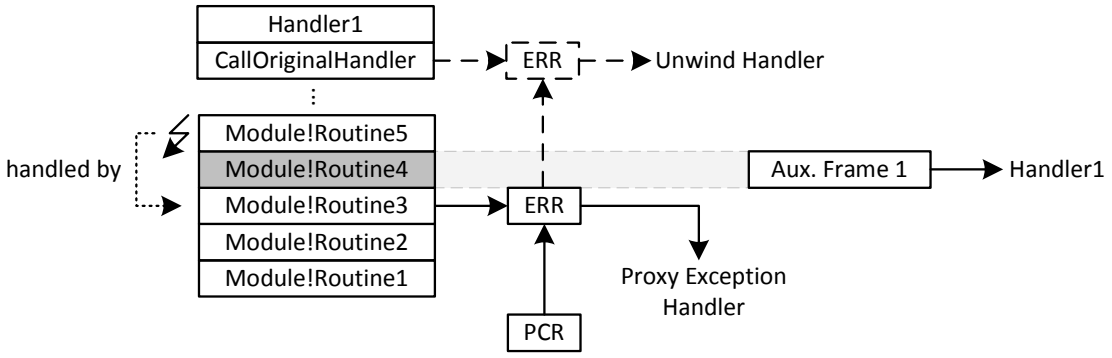

Figure 7.4: State during execution of the original exception handler

Figure 7.4 illustrates the state at the point where the proxy exception handler has delegated the call to the original handler: The SEH chain now temporarily contains an additional registration record, illustrated with dashed lines.

Revisiting the situation discussed before, the behavior of `RtlUnwind` will now change. Although the SEH record installed by Routine3 has been the top record when the exception occurred, it is not the top record any more when the exception is dispatched. Noticing this, `RtlUnwind` will now unwind the top record, which involves calling the unwind handler: The auxiliary stack can now be properly unwound.

7.6.3.2 Non-Topmost Exception Handler Initiating an Unwind

The situation slightly changes when more than one exception registration is involved and not the topmost, but rather a handler of one of the bottom registration records agrees to handle the exception. This situation is illustrated in figure 7.5: Routine2 and Routine4 have been instrumented; Routine 5 raises an exception, Routine2 will finally handle the exception. The handler of the topmost exception registration record (corresponding to Routine4), which is the proxy exception handler, is invoked first. Like in the previous example, it will set up a temporary registration record (not shown) and will delegate the call to Handler2. However, Handler2 declines to handle the exception and returns `ExceptionContinueSearch`.

Proceeding with the next exception registration record, the proxy exception handler is called again. This time, however, a different auxiliary stack frame applies and the call will be delegated to Handler1. Again, a temporary registration record is set up before invoking the handler (shown with dashed lines). Handler1 agrees to handle the exception and initiates an unwind by calling `RtlUnwind`.

As indicated before, `RtlUnwind` will walk the list of registration records once more. That is, it will first invoke the handler associated with the topmost registration record, which, again, is the proxy exception handler.

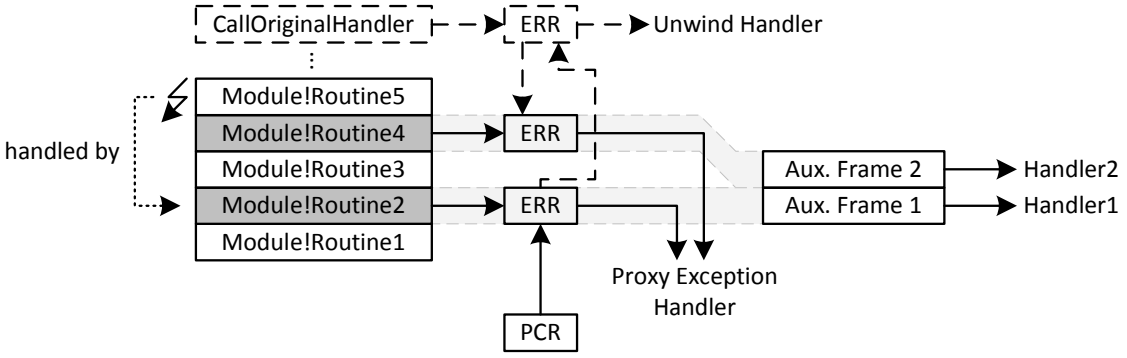

Figure 7.5: Bottom handler handling the exception

Unlike the previous call, however, the handler is requested to perform unwinding. In this case, it will pop off the topmost auxiliary stack frame, and will delegate the call to Handler2.

The next registration in the chain is the temporary registration installed before Handler2 was invoked. Its handler, the unwind handler, is invoked and will pop off the last outstanding auxiliary stack frame, which is Frame 1. At this point, unwinding has been completed and the auxiliary stack has properly been unwound.

7.6.3.3 Multiple Instrumented Routines Sharing a SEH Record

So far, only situations have been considered where the number of stack frames of instrumented routines, and thus the number of auxiliary stack frames, matched the number of SEH registration records on the stack. When more than one auxiliary stack frame maps onto a single SEH registration record, a slight variation of the scheme is required, as figure 7.6 suggests.

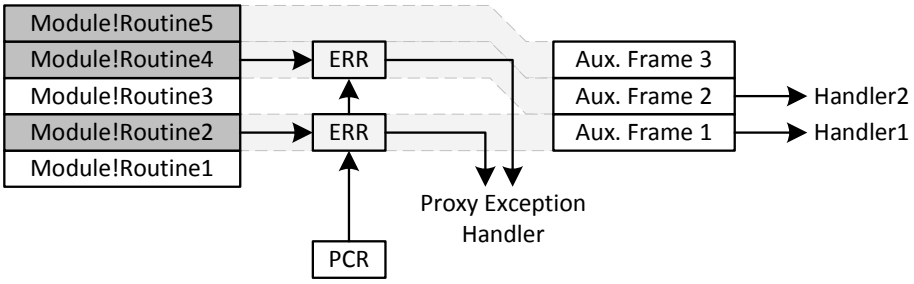

Figure 7.6: Multiple auxiliary stack frames mapping onto a single exception registration record

If the respective registration record is found to already point to the proxy exception handler (as in the case of Routine5), the handler is not replaced once again. Rather, the auxiliary stack unwinding logic accounts for such situations: Instead of popping only the topmost frame, a call to the unwind handler or the unwind part of the proxy exception handler causes all frames up to and including the first frame containing a pointer to an original exception handler routine to be popped. In the specific case depicted by figure 7.6, this means that the auxiliary stack frames 2 and 3 would be unwound at once, while frame 1 is unwound in isolation.

7.6.3.4 Empty SEH Chain

Another situation that has been ignored so far is the possibility that no SEH record has yet been installed when a traced routine is called. That is, the respective pointer in the PCR contains the special value `EXCEPTION_CHAIN_END`.

In such situations, the scheme as discussed so far is not applicable. It is, however, also not necessary to install an additional exception handler in this case: If no SEH record has been installed and one of the functions indirectly called by the traced routine raises an exception while still no SEH frame has been set up, this exception will necessarily be left unhanded and lead to a bugcheck. The fact that an auxiliary stack frame has been leaked is in this case of no real concern as the system is about to stop anyway.

As a consequence, if the PCR is found to not have a single exception registration record registered, the entire process of installing an additional exception handler can safely be skipped.

Finally, it is worth mentioning that the entire implementation is SafeSEH-conforming.

7.7 Event Handling

Besides the ability to capture events such as procedure entry and exit, a vital component of a tracing solution is the handling of such events. Being in charge of recording, buffering and persisting potentially large volumes of event information, event handling also plays a critical role for the performance of a tracing solution.

Although exact numbers depend on the number of routines instrumented, tracing on the level of routines must be assumed to generate significant amounts of data. Especially when tracing is enabled for longer periods of time, storing event information in memory must therefore be assumed to be too expensive in terms of resource consumption. Persisting the information to secondary storage is clearly favorable.

Writing event information synchronously to disk from within the event callback routines invoked on entry or exit of an instrumented routines can be assumed to be both prohibitive for performance and reentrance reasons. An asynchronous approach – temporarily storing the information in memory and having a background thread periodically write the data back to disk – promises to be advantageous in both counts.

One implementation approach could thus be to collect event information in kernel mode memory and having a user mode program collect the information data via appropriate system calls or shared memory. Once retrieved, the program would write the data to a trace file.

Utilizing the Windows NT kernel mode I/O API, however, it is possible for a kernel mode component to directly access and write files, i.e. without having the data be relayed by a user mode program. Requiring less system calls and less memory to be copied, this approach can clearly be expected to be more efficient and has thus been chosen for NTrace.

7.7.1 Buffer Management

A variety of buffer management strategies exist for such asynchronous tracing approaches. Not aiming to be exhaustive, a list of three basic approaches shall be briefly discussed:

- Maintaining buffer space shared among all processors and threads.
- Processor-private buffer space – each processor is assigned buffer space that may only be used by code running on this processor.

- Thread-local buffer space – each thread is associated private buffer space.

Regarding synchronization, reentrance, ordering and anticipated cache behavior, each of these approaches has its individual advantages and shortcomings.

7.7.1.1 Synchronization

Synchronizing access to buffer space among concurrently executing threads is required for the approach utilizing a globally shared buffer. For processor-local and thread-local buffer space, concurrent access cannot occur and synchronization is not required.

7.7.1.2 Reentrance

Although concurrent access cannot occur for the latter two approaches, they are, like the first approach, prone to reentrance-related issues. Regardless of the individual implementation, allocating and writing an entry to buffer space can be assumed to be non-atomic, i.e. requiring multiple CPU instructions.

This non-atomicity can become a problem as soon as interruptions occur. Unless the IRQL prevents it, interrupts, and, indirectly, *Deferred Procedure Calls* (DPC) as well as *Normal* and *Special Kernel Asynchronous Procedure Calls* (APC) can interrupt code at any time. Running on the same processor/thread, these routines could either themselves be instrumented or may call instrumented routines. In either case, when the interrupted routine has been in the midst of writing an entry to buffer space when it was interrupted, reentrance will occur as soon as the interrupting routine attempts to write its first record to buffer space.

In order to protect against corruption of such thread- or processor local resources, it is thus vital for the respective code to properly deal with reentrance.

7.7.1.3 Ordering

In order to be of worth, event information as stored in buffer space and trace file must obey a defined order. Only if ordering of events can be assumed, it is possible to derive further information such as caller/callee-relations.

Whether total ordering of events across all processors and threads is required or not depends on the individual application. If events such as context switches or lock acquisitions are traced, which have a global impact, total ordering of events may be indispensable.

Routine calls are inherently thread-local, although for certain applications, seeing which routines were executed concurrently may depict valuable information. For the majority of use cases, however, it may be expected that threads will be inspected in isolation and partial ordering, in which only events originating from the same thread are ordered w.r.t. each other, is in fact sufficient.

If total ordering is required, using globally shared buffer space is the natural choice. By the use of other means such as sequence numbers, it is, however, also possible to maintain total order if separate buffer spaces are used, as in the case of CPU- or thread local buffers. Yet, the latter two schemes may be expected to achieve their maximum efficiency when only partial ordering is required.

7.7.1.4 Cache Behavior

If instrumented routines are called at a high frequency, the memory occupied by buffer space can become a *hot* resource. While this is not problematic per se, it can lead to inefficiencies on multiprocessor machines when the buffer space is shared among processors.

If events are produced on several processors at roughly the same rate, certain cache lines will repeatedly be updated from different processors in an alternating fashion. This, however, can lead to bus contention and pipeline stalls, which in turn will induce a non-negligible performance overhead [FP02]. Thread- or processor-local caches can be expected to exhibit more beneficial cache behavior in this regard.

7.7.1.5 Implementation Choice

Concluding the brief discussion of buffering approaches, it is clear that each approach has its individual strengths and weaknesses and neither is capable of suiting all needs. As such, the choice of a buffering approach has not been hardwired in the core library. Rather, the user of this library, which, in the architecture laid out, is the *Kernel Function Boundary Tracing Agent*, decides on the buffering scheme to implement.

The WRK/WMK version is capable of leveraging the WMK infrastructure for event buffering, which internally uses globally shared buffer space [SS07]. Not only will total ordering of events be preserved in this case, the function boundary tracing events will also be ordered with respect to the other events captured by the WMK such as context switches.

The version of the agent targeting the retail Windows kernel uses a custom buffering scheme, following the approach of using thread-local buffer space providing partial ordering. When the agent is loaded, a certain amount of equally-sized buffers is allocated from the non paged pool and are kept in a global list, the *free list*. Typical buffer sizes range between 64 and 256 KB.

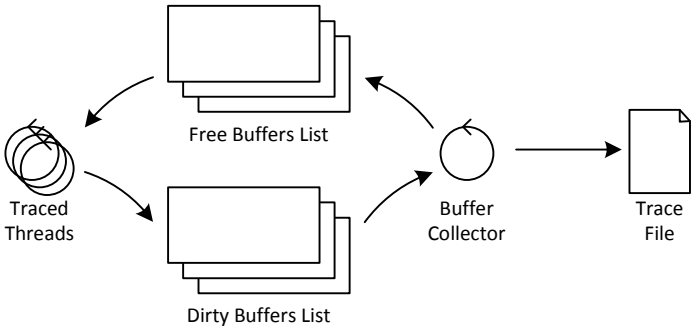

Figure 7.7: Buffer Management Dynamics

Figure 7.7 illustrates the key idea of the implementation. As soon as an event is to be written to buffer space, a buffer has to be obtained from the free list. Using atomic operations, a buffer is removed from this list and attached to the current thread, where it will be used until its space is exhausted. If the next event is to be stored, the existing buffer is put on another global list, the *dirty list* and a new, empty buffer is obtained from the free list.

Periodically, the *buffer collector*, a dedicated system thread, will take each buffer from the dirty list and flush its content to the trace file. After reinitializing the buffer, it is re-added to the free list.

A number of aspects of this approach are worth highlighting. Exclusively relying on interlocked *SList* functions, the entire buffer management implementation can be characterized as being lock-free. Regarding cache behavior, the implementation limits interaction with global memory to obtaining and releasing buffers. As one buffer is capable of holding several thousand events, this interaction occurs on a less frequent basis. Finally, using

thread-local buffers allows storing certain information such as thread and process identifiers once per buffer, rather than once per event record, which in turn helps cutting the size of event data.

Finally, protection against reentrance issues is provided by the mechanism implemented by the CallProxy and EntryThunk. This mechanism has been discussed in section 7.5.2.

7.7.2 Timing

In order to allow making judgments about timing behavior, trace events may include a timestamp reflecting the time at which the routine entry or exit event was captured. Based on these timestamps, the time elapsed during the invocation of a routine can be easily calculated.

To be effective, the timer used for taking these timestamps should satisfy at least the following requirements:

- Taking a timestamp should not induce a significant performance impact.
- Time should advance monotonically and at a constant rate.
- As many routines can be expected to complete in significantly less than 1 millisecond, the timer resolution should be below 1 millisecond.
- Time as observed by different processors should be synchronous. Alternatively, the clock skew should be bounded to a value small enough to not have a significant impact on the correctness and value of time measurements.

Not aiming to be exhaustive, the following list shows the prevalent sources for timing information offered by the Windows NT kernel API:

- System Time, available via `KeQuerySystemTime`.
- Tick Count, available via `KeQueryTickCount`.
- Interrupt Time, available via `KeQueryInterruptTime`.
- High Resolution Performance Counter, available via `KeQueryPerformanceCounter`.
- Bypassing the API, `rdtsc` can be used to obtain the *Time Stamp Counter* (TSC).

The first three sources all provide a resolution in the order of 10 milliseconds and are thus unsuitable for obtaining the desired timestamps. `KeQueryPerformanceCounter` supports fine-grained resolution, yet, the exact behavior of this routine depends on a number of factors. `KeQueryPerformanceCounter` is part of the *Hardware Abstraction Layer* (HAL) and its implementation differs between the uniprocessor and multiprocessor HAL. Moreover, it may use different time sources depending on hardware configuration. Short of public documentation on this topic, variations among operating system releases should also be expected.

Potential time sources used by `KeQueryPerformanceCounter` include the *ACPI timer* (also referred to as PM clock), `rdtsc`, and the *8254 Programmable Interval Timer*[2]. As of Windows Vista, the *High Precision Event Timer* (HPET) is also supported.

[2]Note that this information has been obtained from disassembly as no authorative information on this topic seems to be available to date.

Although `KeQueryPerformanceCounter` provides a clean abstraction of the inhomogeneity among hardware, this variance makes it hard to judge the applicability of the function for the specific usage scenario. Assuming the 8254 timer is not being used, `KeQueryPerformanceCounter` can be expected to execute quickly and provide sufficient precision. However, especially when the TSC is used (either via `KeQueryPerformanceCounter` or directly by using `rdtsc`), it is crucial to notice that synchronicity is not guaranteed.

The TSC is maintained on a per-processor basis and is guaranteed to advance monotonically [Int07d]. It is, however, crucial to notice that due to power management, significant skew between the TSC values of different processors of a multiprocessor system may emerge [Bru05].

If a thread is migrated from one processor to another while in the midst of performing time measurements, this clock skew can easily render the timing measurements performed by this thread meaningless. Worse yet, it is possible that the TSC of the second processor is behind the TSC of the first processor, so that time has effectively went backwards for the thread having migrated.

Unless such processor migrations are explicitly prohibited by using thread-affinity, neither synchronicity nor monotonicity may thus be assumed for the TSC on a multiprocessor machine.

A detailed discussion of the advantages and disadvantages of the various timers and strategies to deal with clock skew is beyond the scope of this work. Despite its lack of guarantees on multiprocessor machines, both `KeQueryPerformanceCounter` and `rdtsc` have been used in practice for providing the timestamps. While achieving good results on uniprocessor machines, the implication on multiprocessor machines is that timing results become flawed as soon as TSC skew emerges.

7.7.3 Symbols

The most vital part of an event record is the information which routine has just been entered or left and is thus the cause of the event. Hence, each event record contains the *virtual address* (VA) of its corresponding routine.

Alternative options to handle this information would have been to use the *relative virtual address* (RVA), i.e. the offset of the routine relative to the containing module's load address or the name string of the routine. The latter option would necessitate a certain amount of symbol handling or maintenance of a mapping between addresses and names to be managed in kernel space, which was undesired. Moreover, storing names requires significantly more buffer space, which is a scarce resource.

Calculating the RVA requires a non-negligible amount of processing as well – which is unfortunate for three reasons: Firstly, it slows down a code path that must be expected to be executed at high frequency. Secondly, this processing may have to occur at elevated IRQL – calling certain APIs to perform the calculation may thus not be feasible. Finally, the additional processing would increase the danger of reentrance, as the API routines called during the conversion process may themselves be instrumented.

The burden of converting from VA to RVA and finally to a name is thus put on the application reading the trace file. For this to work, the trace file contains two critical pieces of information for each module for which routines have been instrumented: The load address, which is required for conversion between VA and RVA, and the debug information of the module, which is part of the module's PE image and is required for identifying the symbol files exactly matching the module. Given this information, it is possible to read a trace file and properly resolve all symbol information – even in the case where the operating system release and file versions differ between the machine having produced and the machine reading the trace file.

7.7.4 Call Nesting

The event records stored in the trace file should allow reconstructing the call flow and the nesting of calls on each thread. For this to work, pairs of entry and exit event records must be found that both correspond to a single routine execution. As event records are written sequentially, finding these pairs should be possible solely based on their order in the trace file (respecting the fact that entries may origin from different threads). Figure 7.8 illustrates this idea by showing a trace of a very simple program: A function *Main* calls *Foo*, which in turn calls *Bar*.

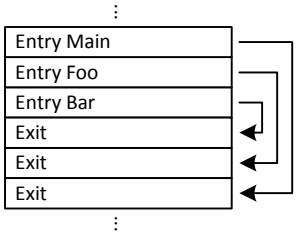

Figure 7.8: Finding pairs of event records

This approach, however, fails as soon as events have to be dropped due to resource constraints. When buffer space for storing event records is depleted and no empty buffer space can be obtained, events have to be dropped. Picking up the previous example and assuming that the exit event record for *Bar* could not be written due to memory shortage, the resulting trace looks as illustrated in figure 7.9.

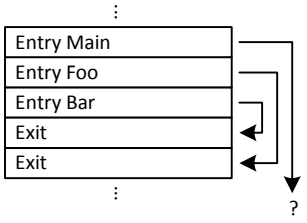

Figure 7.9: Finding pairs of event records in case of lost events

The fact that the count of entry and exit event records does not match any more indicates exit event losses. Yet, based on this data, it is not possible to decide on which exit event has in fact been lost. Not being able to properly match entry- and exit events any more thwarts any attempts to correctly reconstruct nesting relationship among calls.

To avoid such situations from occurring, the implementation approach differs in that all exit events include the routine address. As this address is not available any more during call postprocessing, it is preserved as part of the auxiliary stack frame.

Having the routine address included in exit event records now allows sanity checks to be made when reading the file. If a pair of entry and exit events does not have matching addresses, it is safe to assume that at least one entry or exit event must have been lost between these two records. To a certain extent, this even allows implying which events must have been lost. Being able to simulate such events significantly improves the quality of resulting trace analysis.

7.8 Concluding Remarks on Runtime Code Modification

Employing runtime code modification, it is crucial for NTrace to properly address the challenges of runtime code modification that have been discussed in chapter 4. Although the mechanics used to overcome these challenges have already been considered in the previous sections, they are worth being summarized.

Due to the focus on the IA-32 architecture and Windows NT using the flat memory model, no special considerations regarding the memory model are necessary. Concerning memory protection, however, NTrace has to deal with wrtite-protected portions of images. As discussed in section 7.1.2, additional virtual address mappings are used to circumvent such protection mechanisms.

NTrace also generates code during runtime – namely, the jump necessary to divert execution from the original routine to the trampoline as well as the trampoline itself. Yet, this code is used to override existing code, i.e. it is written to memory which can be assumed to grant *execute* permissions already. As such, no special measures have to be taken in order to comply with the restrictions of Data Execution Prevention.

As discussed in section 6.2, the jump used to divert execution to the trampoline uses a fixed jump distance. The distance of the jump from the trampoline to the CallThunk, however, varies depending on the location of the routine being instrumented and the load address of the FBT agent driver. However, as both origin and target of the jump fall into kernel virtual address space, the distance is guaranteed to be smaller than 2 GB in size. As 2 GB also is the maximum distance supported by the near jump instruction used, jump-distance related issues are avoided.

Regarding cross-modifying code and atomicity, NTrace follows the guidelines defined by Intel [Int07c]. The implementation of the respective algorithm, which is based on the usage of DPCs, has been discussed in section 7.1.2. Yet, usage of this algorithm alone does not protect against the issues of concurrent execution (section 4.4.2) as well as preemption and interruption (section 4.4.3).

During instrumentation, these issues are avoided by the fact that NTrace only replaces a single instruction (the `mov edi, edi`) by an equally sized instruction. As the padding area which is used to place the trampoline has to be considered dead code at this point, it is not exposed to similar issues.

For uninstrumentation, the situation is slightly more complex, as two instructions have to be replaced – the short jump with which the `mov edi, edi` has been overwritten and the near jump located in the trampoline. However, as instruction boundaries remain intact, the only additional condition that has to be regarded is the following:

- Thread A is interrupted or preempted while running an instrumented routine. The thread has successfully executed the initial jump to the trampoline, but has not yet run the jump instruction defined by the trampoline itself.
- Thread B performs an uninstrumentation of the respective routine.
- Thread A is resumed and will run whichever code now resides in the padding area.

This situation, however, could be mitigated with the help of a *nop sled*. When the trampoline is revoked, i.e. the jump instruction is removed from the padding area of the respective routine, the freed space is filled with `nop` instructions. Regarding the previous situation, Thread A would, after resuming, in this case safely execute five nop instructions before re-entering the routine.

If, however, the respective routine is instrumented again while one of the threads is still running the nop sled, this algorithm would in fact not be not safe any more. It is therefore more beneficial to use a single forward near jump instruction rather than five nop instructions so that instruction boundaries remain intact.

Due to the nature of hotpatchable images and runtime code modification being limited to specific instructions only, a situation where a basic block boundary is accidently overwritten is avoided from occurring.

NTrace does not employ stack walking in the sense of inspecting the call stack. For instrumentation, such checks are not necessary as instrumentation can be performed regardless of the instruction pointers and return addresses used by other threads. For uninstrumentation, only the auxiliary stack, but not the call stack is inspected – a mechanism that has been discussed in section 7.4. NTrace therefore is not affected by the safety concerns regarding Stack Walking.

Finally, Disassembly as employed by NTrace is limited to validating the instrumentability of a routine. For these checks, it is merely necessary to perform simple memory comparisons. Any of the disassembly-related concerns such as the challenge of discerning code from data as well as dealing with variable instruction lengths do not apply.

Part III

Analysis

8 Performance Measurements

Performance is an important factor for the applicability of a dynamic tracing system. In order to allow assessing the performance properties of NTrace in more detail, a number of measurements will be discussed in the following sections.

Tracing the execution of a software system constitutes additional work that has to be performed. For a tracing system aiming to exhibit decent performance, this additional work, the *runtime overhead*, clearly should be as small as possible and therefore depicts the prevalent figure for performance assessment. Referring back to the criteria discussed in section 2, the runtime overhead can therefore be seen as determining the frugality of the system.

Another aspect closely related to runtime overhead is scalability, i.e. the question how the performance and overhead evolves when the number of instrumented routines, or the number of processors rises.

To address these two aspects of performance, a simple benchmark has been developed and performed.

8.1 Benchmark

The total overhead of the tracing system is influenced by a number of factors. Among them are the total number of routines instrumented, the rate at which instrumented routines are hit and the average length of an instrumented routine. Without taking such factors into consideration, the overhead cannot be quantified properly.

Rather than striving at a universal estimate of the runtime overhead, the overhead is hence measured in relation to the number of events generated for a given workload. This figure is indirectly influenced by the number of routines instrumented and their hit rate.

To gather meaningful data, the system has been observed while being occupied with a large build job. This workload has been chosen for generating a significant amount of I/O – and thus system calls – as well as involving frequent process startups and tear downs. The build job consisted of performing a full rebuild of the Windows Research Kernel sources. Five builds were performed during each run.

To give an impression of the runtime characteristics of this workload, figure 8.1 shows the distribution of traced routines, grouped by prefixes, for a single WRK build with full instrumentation on the kernel image.

8.1.1 System

All test runs have been performed on a Windows Server 2003 SP2 system, using a retail kernel (free build). The system has been equipped with an Intel Core 2 Quad Q6600 2.4 GHz Quad-Core Processor, 2 GB of RAM and a single 465 GB SATA hard disk. In order to discern tracing-related disk accesses from other disk accesses, a dedicated hard disk partition has been used for storing the trace files.

The system ran without kernel debugger attached and Driver Verifier disabled for all drivers. After each test run, the system was rebooted. For the tracing solution itself, the free build, i.e. a non-debug, optimized build was used for all tests. Finally, a total of 512 buffers, each roughly 256 KB in size, has been used for the internal buffer management.

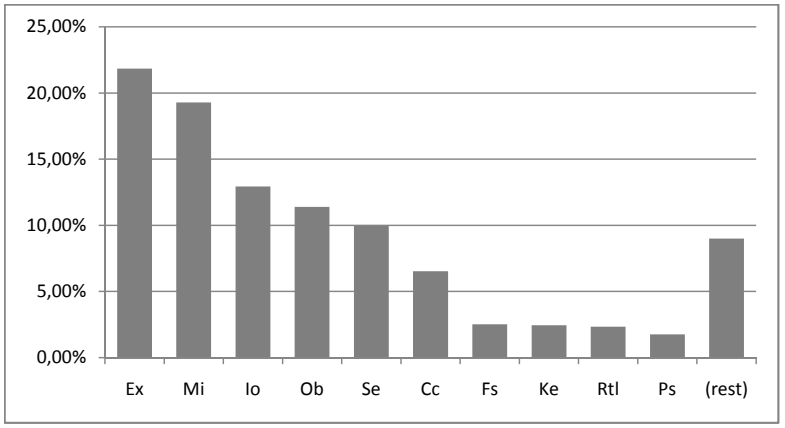

Figure 8.1: Distribution of traced routines for a single WRK build

8.1.2 Performance Counters

In order to observe the test runs in more detail, performance counters were used. The FBT Layer DLL has been augmented to serve as a performance counter DLL, retrieving performance data from the FBT Agent. The supported performance counters are:

- Instrumented Routines Count: The current number of instrumented routines.
- Buffers (Free): Number of free event data buffers in pool.
- Buffers (Dirty): Number of dirty event data buffers waiting to be flushed to disk.
- Buffers (Collected): Number of event data buffers written to disk.
- Prealloc. Pool (Free): Number of free preallocated ThreadData structures.
- Prealloc. Pool (Failed Allocations): Failed allocations from the ThreadData preallocated pool.
- Reentrant executions (Delta): Number of times an event could not be captured because of reentrance (See chapter 7.5.2)
- Events dropped (Entry, Delta): Dropped entry events because of buffer depletion.
- Events dropped (Exit, Delta): Dropped exit events because of buffer depletion.
- Events dropped (Unwind, Delta): Dropped exception unwind events because of buffer depletion.
- Image Infos dropped (Delta): Dropped image information chunks because of buffer depletion.
- Failed chunk flushes (Delta): Number of 'chunks', i.e. packets of trace data that failed to be flushed to disk.
- Events captured: Total number of entry, exit, and unwind events captured[1].
- Events captured (Delta): Entry, exit, and unwind events captured during last sampling interval.
- Exception unwindings: Total number of auxiliary stack unwindings. See chapter 7.6.3.
- Thread Tear downs: Number of threads having exited, requiring a ThreadData tear down.

The FBT Agent delivers absolute figures only. All delta values are calculated by the performance counter infrastructure based on these absolute figures.

Figure 8.2 shows an example screen shot of the Windows *Performance Monitor* displaying data from one of the recorded counter logs.

[1]Note that the values are exact to multiples of 1000 only. To avoid cache line thrashing, events are counted thread locally. Not before 1000 events have been captured on a given thread is the global counter updated. Therefore, the last three digits of the global counter value are always zero.

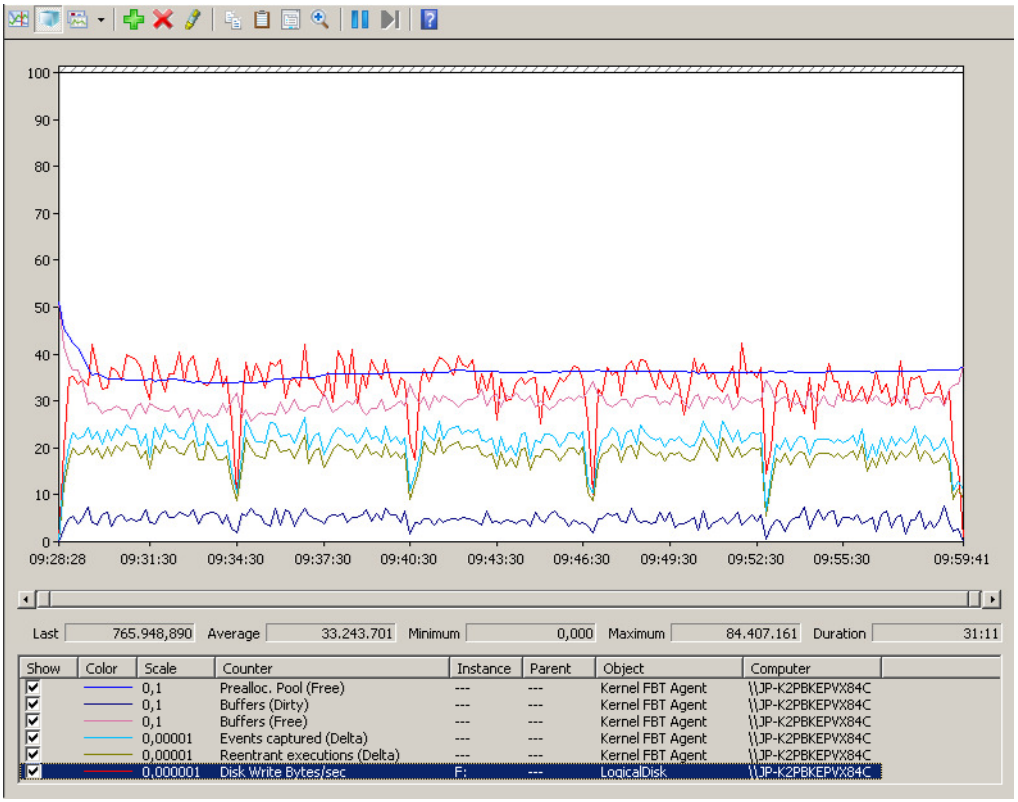

Figure 8.2: Performance Monitor showing selected performance counters from a test run comprising five WRK builds with instrumentation on nt!* (i.e. full instrumentation)

All these performance counters, along with the Windows-provided counters *Logical Disk\Disk Write Bytes/sec* (for the partition containing the trace file) and *Processor\% Processor Time* have been collected for all testruns using a *Counter Log* with a sampling interval of 1 second.

As retrieving the counter data from the FBT agent involves additional I/O processing that may influence the overall results, all counters have also been recorded for the baseline measurements.

The process of managing performance counters and instrumentation as well as starting the build job and measuring elapsed time has been automated in order to avoid unnecessary impact of manual intervention. Still, due to the existence of other processes and numerous background tasks on a Windows system, the results must be expected to reproducible to a certain degree only.

Concurrently running processes such as services must be expected to routinely perform system calls as well. Such system calls, however, may invoke instrumented routines and may thus impact the performance counter statistics. Given that the timing behavior of the various processes is unknown, their behavior must be expected to vary among different test runs. As a result, the exact performance counter results must be expected to vary as well. Still, this impact is assumed to be of minor significance and is thus ignored in the following discussion.

8.1.3 Test runs

Three types of test runs have been conducted:

- Kernel, capturing only: The kernel image has been instrumented to various degrees and the events have been captured. Once captured, however, the events were dropped

immediately. That is, events have not been written to a buffer and have not been written to disk. The results from this run therefore show how much overhead the capturing itself introduces.

- Kernel, with writing to disk. Again, the kernel image has been instrumented to various degrees. Events were handled by the buffer management and finally written to disk. The results from this run therefore show the overall overhead of both capturing and persisting the event data.
- Driver, with writing to disk. Not the kernel, but a driver has been instrumented. Having chosen ntfs.sys, the driver implementing the NTFS file system, as target, the rationale behind this test run is merely to show that certain drivers may yield substantially different figures, especially with regard to exception unwinding.

Handling and writing event data to disk clearly imposes additional overhead, so that the total overhead in the respective test runs will necessarily be higher. There is, however, an additional point to notice: The code used to perform the task of writing event data to disk itself makes use of kernel routines such as those of the I/O manager. In some test runs, these routines have been instrumented, so that writing events to disk generates additional events. Handling these additional events, however, will again increase the workload and lead to a higher total overhead.

In those test runs where event data is flushed to disk, the buffer management system discussed in section 7.7.1.5 is used. Due to its asynchronous nature and its, albeit limited and *lock-free*, synchronized use of globally shared resources, a slightly increased degree of variability and non-determinism should also be expected in the runtime behavior and timing measurement results.

To observe the behavior of the system for different degrees of instrumentation, ten sets of routines to be instrumented have been compiled. However, as not the sheer number of routines instrumented but rather the number of events generated is significant for performance measurements, these sets have not been chosen based on their size but on the anticipated number of events generated.

The first set contains all instrumentable routines of the kernel image. Based on the number of events generated by a WRK build using this instrumentation set, and taking the distribution among routine prefixes illustrated in figure 8.1 into account, nine additional routine sets have been defined. Comprising only a subset of the instrumentable routines, each set has been chosen so that it roughly generates a certain fraction of the maximum number of events. Table 8.1 lists these sets, along with their estimated percentage of events generated in relation to full instrumentation.

8.1.4 Results

The following three tables show the raw results of the test runs performed.

Based on these raw numbers, a number of basic observations can be made:

- Although the workload has been the same for all runs, the figure *Thread Tear downs* varies slightly. This must be expected to be a result of the existence of background tasks and other activity on the system.
- Comparing the *Time* and *Events/sec (non reentrant)* figures of table 8.2 with those of table 8.3, shows that the effort required to handle and write the event data to disk is non negligible: Builds take significantly longer and as a consequence, the event rate is lower when events are written to disk.

Table 8.1: Instrumentation Sets

Name	Set (Prefixes)	Estimated output (relative to full instrumentation)	Number of routines
S_0(*baseline*)	(none)	0%	0
S_{10}	Se	10%	182
S_{20}	Io	20%	685
S_{30}	Io, Ob	30%	813
S_{40}	Ex, Mi	40%	681
S_{50}	Ex, Mi, Ob	50%	809
S_{60}	Ex, Mi, Io	60%	1365
S_{70}	Ex, Mi, Io, Se	70%	1546
S_{80}	Ex, Mi, Io, Se, Ob	80%	1674
S_{90}	Ex, Mi, Io, Se, Ob, Fs, Cc	90%	1995
S_{100}	*	100%	5131

Table 8.2: Measurements: Kernel, capturing only

Set	Time [ms]	Trace size [MB]	Events/sec (non reentrant)	Events total (non reentrant)	Thread tear-downs	Excep. Un-winds	Reentran-cies	Logical Disk [KB/sec]
S_0	397,941	0	N/A	N/A	N/A	N/A	N/A	N/A
S_{10}	442,263	N/A	899,254	398,381,000	1,611	0	4	N/A
S_{20}	467,365	N/A	1,359,319	636,293,000	1,630	0	978	N/A
S_{30}	515,550	N/A	2,106,662	1,087,088,000	1,647	0	516	N/A
S_{40}	537,817	N/A	1,714,889	922,708,000	1,672	0	710,437	N/A
S_{50}	592,178	N/A	2,286,698	1,356,510,000	1,649	0	932,404	N/A
S_{60}	603,683	N/A	2,487,878	1,505,372,000	1,625	0	951,276	N/A
S_{70}	669,022	N/A	2,909,408	1,949,586,000	1,668	0	1,176,915	N/A
S_{80}	728,015	N/A	3,288,097	2,397,390,000	1,671	0	1,322,195	N/A
S_{90}	761,557	N/A	3,416,100	2,607,018,000	1,683	0	1,400,906	N/A
S_{100}	884,868	N/A	3,290,311	2,915,831,000	1,703	1	2,702,529	N/A

Table 8.3: Measurements: Kernel, with writing to disk

Set	Time [ms]	Trace size [MB]	Events/sec (non reentrant)	Events total (non reentrant)	Thread tear-downs	Excep. Un-winds	Reentran-cies	Logical Disk [KB/sec]
S_0	397,941	0	N/A	N/A	N/A	N/A	N/A	N/A
S_{10}	497,419	6,143	802,811	398,423,000	1,610	0	4	12,943
S_{20}	660,023	10,030	994,044	651,895,000	1,643	0	53	15,900
S_{30}	884,267	17,168	1,269,215	1,115,821,000	1,688	0	454	20,297
S_{40}	783,479	16,967	1,407,650	1,100,520,000	1,665	0	603,830	70,252
S_{50}	1,064,575	25,317	1,544,235	1,649,048,000	1,686	0	653,136	24,703
S_{60}	1,026,918	28,464	1,869,531	1,919,400,000	1,695	0	750,117	28,953
S_{70}	1,453,399	37,063	1,689,525	2,418,062,000	1,681	0	712,852	26,273
S_{80}	1,648,374	45,775	1,880,908	2,987,434,000	1,715	0	789,757	28,368
S_{90}	1,566,561	52,347	2,181,138	3,416,633,000	1,677	0	937,571	28,369
S_{100}	1,869,358	60,773	2,134,650	3,967,796,000	1,677	0	1,824,291	33,243

- Regarding the number of reentrances, there is a sharp rise between S_{30} and S_{40} (see figure 8.3). S_{30} includes `nt!Io*` and `nt!Ob*`, S_{40} comprises `nt!Ex*` and `nt!Mi*`. It is the latter set of routines, `nt!Mi*`, that may be expected to be the cause of this rise: These routines, part of the Memory Manager, are frequently invoked from within trap and interrupt handlers. As such, reentrance is more likely to occur in this routine set than it is in other sets.

Table 8.4: Measurements: Ntfs.sys, with writing to disk

Set	Time [ms]	Trace size [MB]	Events/sec (non reentrant)	Events total (non reentrant)	Thread tear-downs	Excep. Un-winds	Reentran-cies	Logical Disk [KB/sec]
S_0	397,941	0	N/A	N/A	N/A	N/A	N/A	N/A
S_{100}	1,085,506	24,011	1,447,219	1,589,072,000	1,482	12,251	1,449,883	23,045

- While exception unwinds are very rare when tracing the kernel itself, ntfs.sys obviously makes frequent use of exceptions and unwinds. With more than 10 unwinds per second in average, this also stresses the importance of the tracing system to properly deal with SEH.

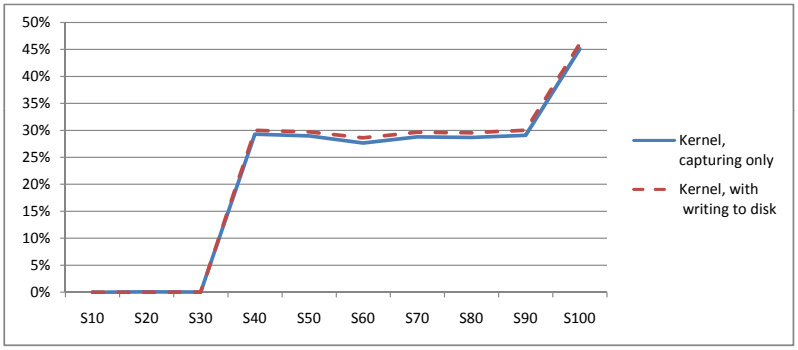

Figure 8.3: Percentage of events dropped due to reentrance

By relating the figures obtained by the various test runs with those obtained by the baseline measurement, it is possible to calculate the overhead imposed by the tracing activity:

- *Total Overhead*: Percentage by which the total run time has increased.
- *Overhead/100 M events*: Overhead in relation to the number of events generated, i.e. percentage by which the total run time has increased for each 100 million events handled. Events dropped due to reentrance are ignored by this figure.
- *Avg. Time/event*: Average time value, in nanoseconds, by which the total run time has increased for each event handled. Again, events dropped due to reentrance are ignored by this figure.

Table 8.5: Measurements: Overhead

	Kernel, capturing only			Kernel, with writing to disk		
	Total Overhead	Overhead/ 100 M events	Avg. Time/ event [ns]	Total Overhead	Overhead/ 100 M events	Avg. Time/ event [ns]
S_{10}	11.14%	2.80%	111	25.00%	6.27%	249
S_{20}	17.45%	2.74%	109	65.86%	10.10%	402
S_{30}	29.55%	2.72%	108	122.21%	10.95%	435
S_{40}	35.15%	3.81%	151	96.88%	8.80%	350
S_{50}	48.81%	3.60%	143	167.52%	10.16%	404
S_{60}	51.70%	3.43%	136	158.06%	8.23%	327
S_{70}	68.12%	3.49%	139	265.23%	10.97%	436
S_{80}	82.95%	3.46%	137	314.23%	10.52%	418
S_{90}	91.37%	3.50%	139	293.67%	8.60%	342
S_{100}	122.36%	4.20%	166	369.76%	9.32%	370

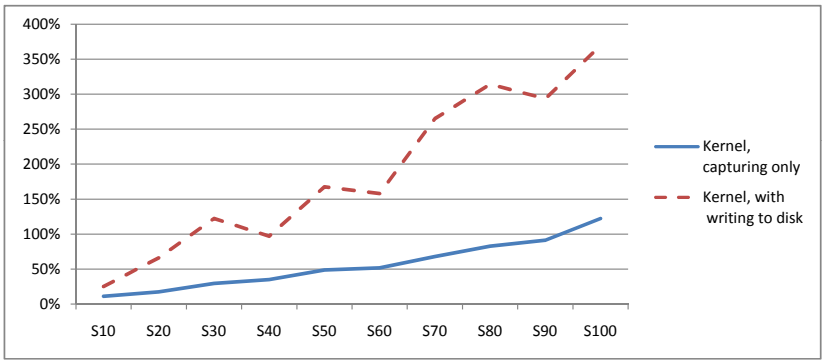

Figure 8.4: Total Overhead

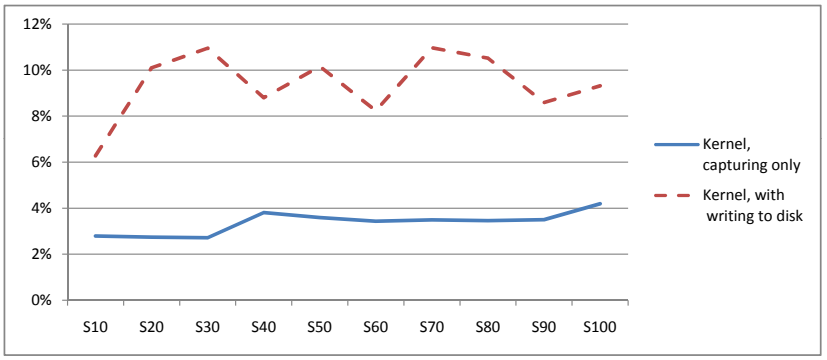

Figure 8.5: Overhead for each 100 million events handled

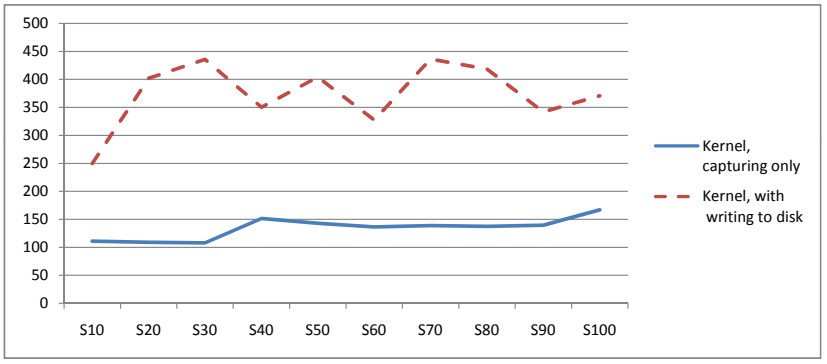

Figure 8.6: Average overhead for each event handled, in nanoseconds

Figure 8.4 shows the *Total Overhead* in relation to different routine sets. With full instrumentation on the kernel image, the build process slows down by roughly 122%. When the event data is also flushed to disk, the overhead increases to 369%.

The fact that the overhead rises with an increasing number of events handled certainly does not come at a surprise. Yet, it is notable that the overhead evolves gracefully, i.e. almost linearly. The diagram also underlines what has been indicated before – that those measurements involving writing event data to disk show higher variability.

However, figure 8.4 does not yet express in how far the overhead of handling a single event is dependent on the total number of events captured. For the solution to be *scalable*, it is crucial that the overhead evolves gracefully when the total number of events increases. That is, the overhead per event should not rise significantly.

The diagrams 8.5 and 8.6 show how the overhead per event evolves. With an increasing number of total events captured, the overhead per single event indeed increases, yet only slightly.

To explain this rise, it is important to notice that the events dropped due to reentrance impose an overhead, yet are not regarded by these numbers. In fact, comparing the respective graphs of these two figures with the those of figure 8.3, there is a remarkable similarity to be noticed: Between S_{30} and S_{40} as well as between S_{90} and S_{100}, there is a steep incline in all graphs.

To yield more exact figures, the overhead imposed by events that have been dropped due to reentrace would need to be factored in. This, however, is problematic to achieve as the overhead imposed by a reentrant event is less than the overhead of a properly captured event. Nonetheless, it is safe to assume that the rise is at least partially caused by reentrance.

Finally, although less smooth, the graphs illustrating the overhead when events are additionally written to disk evolve gracefully in all diagrams as well. This may be interpreted as an indication for that the buffer management and disk flushing activity does not suffer from any serious performance bottlenecks.

9 Conclusion

Dynamic function boundary tracing of kernel mode components has many potential applications, yet its potentials have largely been dormant on the Windows NT platform.

This thesis has presented an overview and has proposed a classification scheme covering techniques that may be used for implementing such tracing systems. With the creation of NTrace, we demonstrated that implementing such a tracing system for Windows NT is in fact technically feasible.

In addition to that, we have shown that it is possible for a tracing system to integrate with Windows Structured Exception Handling. Besides allowing the system to attain exception safety, such integration also adds significant value to the tracing results as it allows exception unwinds, like function entry and exit events, to be properly traced.

Utilizing synergy effects with the Microsoft Hotpatching infrastructure, NTrace also has shown that the challenges of runtime code modification can be successfully overcome.

Although the exact requirements on a tracing solution vary with the individual purpose, the general aims of creating a system that is detailed, frugal, and scalable could hence be achieved.

Limitations and Outlook

The focus of NTrace has clearly lied on capturing trace information while maintaining reasonable performance. As such, the system should primarily be seen as providing the foundation for tools offering features going beyond mere recording of trace information. Such features could include the ability to filter events – such as by limiting the capturing to certain threads only – as well as the capability to record parameter information. Other tools, such as profilers, are equally conceivable.

But there certainly are also aspects to the current implementation of NTrace that leave room to improvement. Currently, the most important limitation can be seen in the relatively high loss rate of events due to reentrace. While the current implementation has proven to be able to avoid reentrace-caused hazards, the *window* of instructions of the CallProxy and EntryThunk in which reentrace leads to events being dropped is still larger than necessary. By optimizing and restructuring the respective assembly code, however, the author expects that the drop rate should be able to be lowered significantly.

Much of the implementation of NTrace is both workable in kernel and in user mode. In fact, the entire FBT core library has been implemented and tested for either mode. As such, creating the additional tool support to add the capability to trace user mode processes seems highly auspicious in order to increase the potential fields of application for NTrace.

Finally, NTrace is currently limited to the IA-32 architecture. Although a non-negligible part of the implementation – such as instrumentation and exception handling – is architecture-specific, porting NTrace to AMD64 seems viable, albeit not trivial. Notwithstanding their common heritage, AMD64 poses different challenges on such an implementation. For instance, while being slightly more forgiving with regard to runtime code modification, the increased address space brings up the issues of jump distances again. Moreover, leveraging hotpatchable images on AMD64 requires a certain, although limited amount of disassembly to be performed at runtime. PatchGuard [Cor07], the facility of AMD64 Windows kernels to prevent certain code modifications at runtime, poses another challenge to such porting efforts. That is, NTrace on AMD64 would likely be limited to tracing drivers, while not being able to trace the kernel itself as well the system images *hal* and *ndis* [Joh05]. Still, such porting effort could further enlarge the applicability of NTrace.

Bibliography

[Arn08] Jeffrey Brian Arnold. Ksplice: An automatic system for rebootless Linux kernel security updates. 2008.

[AVAU88] Ravi Sethi Alfred V. Aho and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[BC05] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[BH99] D. Brubacher and G. Hunt. Detours: Binary Interception of Win32 Functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1999.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[Box98] Don Box. *Essential COM*. Addison Wesley, 1998.

[Bro99a] Keith Brown. Building a Lightweight COM Interception Framework, Part I: The Universal Delegator. Microsoft Systems Journal, Vol 14 No 1, 1999. URL `http://www.microsoft.com/msj/0199/intercept/intercept.aspx`, retrieved 10.04.2008.

[Bro99b] Keith Brown. Building a Lightweight COM Interception Framework, Part II: The Guts of the UD. Microsoft Systems Journal, Vol 14 No 2, 1999. URL `http://www.microsoft.com/msj/0299/intercept2/intercept2.aspx`, retrieved 10.04.2008.

[Bru04] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[Bru05] R. Brunner. TSC and Power Management Events on AMD Processors, 2005. URL `http://lkml.org/lkml/2005/11/4/173`, retrieved 3.06.2008.

[CK94] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[Clo01] Shaun Clowes. Modifying and spying on running processes under linux and solaris, 2001. URL `http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt`, retrieved 17.09.2008.

[CM] Cristina Cifuentes and Vishv Malhotra. Binary translation: Static, dynamic, retargetable?

[Com95] TIS Committee. Tool interface standard (tis) executable and linking format (elf) specification, version 1.2, 1995.

[Cor97] International Business Machines Corporation. OS/2 Warp 4 fixpack 4, readme.dbg, 1997. URL `http://hobbes.nmsu.edu/pub/os2/system/patches/fixpack/warp_4/xr_m004/readme.dbg`, retrieved 17.04.2008.

[Cor06a] Microsoft Corporation. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003 (Knowledge Base Aricle 875352), 2006. URL `http://support.microsoft.com/kb/875352`, retrieved 14.05.2008.

[Cor06b] Microsoft Corporation. Visual Studio, Microsoft Portable Executable and Common Object File Format Specification. Windows Harware Developer Central, 2006. URL `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, retrieved 11.04.2008.

[Cor07] Microsoft Corporation. Patching Policy for x64-Based Systems. Windows Harware Developer Central, 2007. URL `http://www.microsoft.com/whdc/driver/kernel/64bitpatching.mspx`, retrieved 10.04.2008.

[Cor08a] Microsoft Corporation. DbgHelp Reference, 2008. URL `http://msdn.microsoft.com/en-us/library/ms679292.aspx`, retrieved 09.07.2008.

[Cor08b] Microsoft Corporation. Debugging Tools for Windows, 2008. URL `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`, retrieved 09.07.2008.

[Cor08c] Microsoft Corporation. Driver Verifier, 2008. URL `http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx`, retrieved 14.05.2008.

[CSL04] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, 2004.

[Dev07] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2007.

[DMS06] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Avoiding Software Vulnerabilities*. Addison-Wesley, 2006.

[EEL+97] Susan J. Eggers, Joel Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors, September/October 1997.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publiching, 2005.

[FP02] M. Friedman and O. Pentakalos. *Windows 2000 Performance Guide*. O'Reilly, 2002.

[Gar02] Garret J. Buban, V. Donlan, Adrian Marinescu, Thomas D. McGuire, B. Probert, H. Vo, Zheng Wang. Patent 20040107416: Patching of in-use functions on a running computer system, 2002.

[HB05] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison Wesley, 2005.

[Hir05] Masami Hiramatsu. Overhead Evaluation about Kprobes and Djprobe (Direct Jump Probe), 2005. URL `http://lkst.sourceforge.net/docs/probes-eval-report.pdf`, retrieved 17.09.2008.

[HMC94] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, 1994.

[HMG+97] Jeffrey K. Hollingsworth, Barton P. Miller, M. J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. MDL: A language and compiler for dynamic program instrumentation. In *IEEE PACT*, 1997.

[Int02] Intel. *Intel Pentium III Xeon Processor Specification Update*, 2002.

[Int07a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 1: Basic Architecture. Intel Corporation, 2007.

[Int07b] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3B: System Programming Guide, Part 2. Intel Corporation, 2007.

[Int07c] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A: System Programming Guide, Part 1. Intel Corporation, 2007.

[Int07d] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2B: Instruction Set Reference, N–Z. Intel Corporation, 2007.

[Joh05] Ken Johnson. Bypassing PatchGuard on Windows x64. Uninformed, Volume 3, 2005. URL `http://uninformed.org/index.cgi?v=3&a=3&t=sumry`, retrieved 15.08.2008.

[Joh06] Ken Johnson. Anti-Virus Software Gone Wrong. Uninformed, Volume 4, 2006. URL `http://www.uninformed.org/?v=4&a=4&t=sumry`, retrieved 10.04.2008.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[Lem99] Dmitri Leman. Spying on COM Objects. Dr Dobb's Journal, July 1999 Issue, 1999. URL `http://www.ddj.com/windows/184416546?pgno=5`, retrieved 14.04.2008.

[Lem00] Dmitri Leman. Tracing NT Kernel-Mode Calls. Dr Dobb's Journal, April 2000 Issue, 2000. URL `http://www.ddj.com/showArticle.jhtml?articleID=184416246`, retrieved 10.04.2008.

[Mas07] Masami Hiramatsu and Satoshi Oshima. Djprobe – Kernel probing with the smallest overhead. In *Proceedings of the Linux Symposium, Volume One*, pages 189–199, 2007.

[Mic07] Sun Microsytems. *UltraSPARC Architecture 2005*, volume Draft D0.9, Privilege Levels: Privileged and Nonprivileged. Sun Microsystems, Inc., 2007.

[MJ07] Matt Miller and Ken Johnson. A Catalog of Windows Local Kernel-mode Backdoor Techniques. Uninformed, Volume 8, 2007. URL `http://www.uninformed.org/?v=8&a=2&t=sumry`, retrieved 10.04.2008.

[Mol08] Ingo Molnar. what's up for v2.6.25 in x86.git. Linux Kernel Mailing List, 2008. URL `http://kerneltrap.org/mailarchive/linux-kernel/2008/1/21/588524`, retrieved 14.04.2008.

[Moo01] Richard Moore. A Universal Dynamic Trace for Linux and other Operating Systems. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[MPK06] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the guts of kprobes. In *Proceedings of the Linux Symposium, Volume Two*, pages 101–115, 2006.

[Neb00] Gary Nebbett. *Windows NT/2000 Native API Reference*. MTP, 2000.

[Net04] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Trinity College, University of Cambridge, 2004.

[OMCB07] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. Jit instrumentation: a novel approach to dynamically instrument operating systems. In *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 3–16, New York, NY, USA, 2007. ACM.

[PDB99] Sandeep Phadke Prasad Dabak and Milind Borate. *Undocumented Windows NT*. Wiley & Sons, 1999.

[Pea00] David Pearce. Instrumenting the linux kernel. Masterthesis, Imperial College, 2000.

[Pie97] Matt Pietrek. A Crash Course on the Depths of Win32Structured Exception Handling, 1997. URL `http://www.microsoft.com/msj/0197/Exception/Exception.aspx`, retrieved 24.06.2008.

[PKFH02] David J. Pearce, Paul H.J. Kelly, Tony Field, and Uli Harder. GILK: A dynamic instrumentation tool for the Linux Kernel. In *Proceedings of the 12th International Conference on Computer Performance Evalution*, 2002.

[PP06] Andreas Polze and Dave Probert. Teaching operating systems: the Windows case. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 298–302, New York, NY, USA, 2006. ACM Press.

[RC97] Mark Russinovich and Bryce Cogswell. Tracing NT Kernel-Mode Calls. Dr Dobb's Journal, January 1997 Issue, 1997. URL `http://www.ddj.com/184410109`, retrieved 10.04.2008.

[Res03] OSR Open Systems Resources. OSR's IRPTracker – Tracking the Life of an IRP in Detail, 2003.

[Rob03] John Robbins. *Debugging Applications for Microsoft .NET and Microsoft Windows*. Microsoft Press, 2003.

[RR03] K Robbins and S. Robbins. *UNIX Systems Programming*. Prentice Hall, 2003.

[SDA02] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited, 2002.

[SEV01] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[SR04] D. Solomon and M. Russinovich. *Windows Internals*. Microsoft Press, 4 edition, 2004.

[SS07] Alexander Schmidt and Michael Schöbel. Analyzing System Behavior: How the Operating System Can Help. In *Workshop on Applied Program Analysis*, 2007.

[Tam01] Ariel Tamches. *Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels*. PhD thesis, University of Wisconsin-Madison, 2001.

[Thi99] Peter Thiemann. Higher-order code splicing. In *European Symposium on Programming*, pages 243–257, 1999.

[TM99] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[Val07] Valgrind's Tool Suite, 2007. URL `http://valgrind.org/info/tools.html`, retrieved 23.04.2008.

[Var05] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston and Brad Chen. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the Linux Symposium*, pages 49–64, 2005.

[Wys08] Rafael J. Wysocki. Freezing of tasks, 2008. URL `http://www.mjmwired.net/kernel/Documentation/power/freezing-of-tasks.txt`, retrieved 18.04.2008.

I hereby affirm that I have written this master thesis independently, without using any sources other than those stated.

Berlin, 21. October 2008