SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation

Amit Vasudevan **Ramesh Yerraballi**

Department of Computer Science and Engineering
University of Texas at Arlington,
Box 19015, 416 Yates St., 300 Nedderman Hall,
Arlington, TX 76019-0015, USA.
Email: {vasudeva, ramesh}@cse.uta.edu

Abstract

Malware — a generic term that encompasses viruses, trojans, spywares and other intrusive code — is widespread today. Malware analysis is a multi-step process providing insight into malware structure and functionality, facilitating the development of an antidote. *Behavior monitoring*, an important step in the analysis process, is used to observe malware interaction with respect to the system and is achieved by employing dynamic coarse-grained binary-instrumentation on the target system. However, current research involving dynamic binary-instrumentation, categorized into probe-based and just-in-time compilation (JIT), fail in the context of malware. Probe-based schemes are not transparent. Most if not all malware are sensitive to code modification incorporating methods to prevent their analysis and even instrument the system themselves for their functionality and stealthness. Current JIT schemes, though transparent, do not support multithreading, self-modifying and/or self-checking (SM-SC) code and are unable to capture code running in kernel-mode. Also, they are an overkill in terms of latency for coarse-grained instrumentation.

To address this problem, we have developed a new dynamic coarse-grained binary-instrumentation framework codenamed SPiKE, that aids in the construction of powerful malware analysis tools to combat malware that are becoming increasingly hard to analyze. Our goal is to provide a binary-instrumentation framework that is unobtrusive, portable, efficient, easy-to-use and reusable, supporting multithreading and SM-SC code, both in user- and kernel-mode. In this paper, we discuss the concept of *unobtrusive binary-instrumentation* and present the design, implementation and evaluation of SPiKE. We also illustrate the framework utility by describing our experience with a tool employing SPiKE to analyze a real world malware.

Keywords: Security, Malware, Instrumentation

1 Introduction

Malware analysis is a multi-step process combining both coarse- and fine-grained analysis schemes that provide insight into malware structure and functionality. Malware analysis environments (sandboxes) thus require various coarse- and fine-grained analysis tools to facilitate the development of an antidote. The first and a very important step in the analysis process, known as *Behaviour Monitoring*, involves monitoring malware behavior with respect to the system, to glean details which aid in further finer investigations. For example, the `W32.MyDoom` trojan and its variants propagate via e-mail and download and launch external programs using the network and registry. Such behavior, which includes the nature of information exchanged over the network, the registry keys used, the processes and files created etc., is inferred by employing coarse-grained analysis pertaining to process, network, registry, file and other related services of the host operating system (OS). Once such behavior is known, fine-grained analysis tools such as debuggers are employed on the identified areas to reveal finer details such as the polymorphic layers of the trojan, its data encryption and decryption engine, its memory layout etc.

Instrumentation – the ability to control constructs pertaining to any code – is a technique that is used for both coarse- and fine-grained analysis. Constructs can either be pure (functions following a standard calling convention) or arbitrary (code blocks composed of instructions not adhering to a standard calling convention). Control means access to a construct for purposes of possible semantic alteration. While instrumentation is trivial for OSs and associated applications that are open source, the task becomes complicated with deployments that are commercial and binary only. To this end there has been various research efforts which achieve instrumentation at the binary level. They fall into two broad categories: *Probe-based* and *JIT*. Probe-based methods such as Dtrace (Cantrill et al. 2004), Dyninst (Buck & Hollingsworth 2000), Detours (Hunt & Brubacher 1999) etc., achieve instrumentation by modifying the target construct in a fashion so as to enable a replacement function to be executed when the original construct is invoked. JIT methods such as Pin (Luk et al. 2005), DynamoRIO (Bruening 2004), Valgrind (Nethercote & Seward 2003) etc., on the other hand achieve instrumentation by employing a virtual machine (VM).

Both methods however fail in the context of malware. Probe-based methods are not transparent because original instructions in memory are overwritten by trampolines. Most if not all malware are sensitive to code modification and even instrument certain OS services for their functioning and stealthness. Also recent trends in malware show increasing anti-analysis methods, rendering probe-based schemes severely limited in their use. Current JIT schemes on the other hand, though transparent, do not contain support for multithreading and SM-SC code and are unable to analyze code executing in kernel-mode. Also JIT schemes are more suited for fine-grained instrumentation and are an overkill in terms of latency for coarse-grained analysis. This situation calls for a new coarse-grained instrumentation framework specifically tailored for malware analysis.

This paper discusses the concept of *unobtrusive binary-instrumentation*, and presents SPiKE, a realization of this concept that provides unobtrusive, portable, efficient, easy-to-use and reusable binary-instrumentation, supporting multithreading and SM-SC code in both user- and kernel-mode. The instrumentation deployed by SPiKE is *unobtrusive* in the sense that it is completely invisible and cannot be easily detected or countered. This is achieved by employing the virtual memory system coupled with subtle software techniques. The framework currently runs on OSs such as Windows (9x, NT, 2K and XP) and Linux on the IA-32 processors (Intel Corporation 2003) and is *portable* on any platform (processor and OS) that supports virtual memory. The framework achieves *efficient* instrumentation using special techniques such as redirection and localized-executions. SPiKEs API is simple yet powerful making the framework *easy-to-use*. Analysis tools are usually coded in C/C++ using SPiKEs API. The API is platform independent wherever possible allowing tool code to be *reusable* among different platforms, while allowing them to access platform specific details when necessary. To the best of our knowledge, SPiKE is the first coarse-grained binary-instrumentation framework that facilitates the construction of powerful malware analysis tools to combat malware that are increasingly becoming hard to analyze.

This paper is organized as follows: We begin by giving an overview of SPiKE's instrumentation mechanism in Section 2. We follow this with a detailed discussion on design and implementation issues in Section 3. In Section 4, we discuss our experience with one of our tools employing SPiKE to analyze a real world malware. We then evaluate in Section 5, the framework performance and compare it against other schemes. Finally, we consider related work on binary-instrumentation in Section 6 and conclude the paper in Section 7 summarizing our contributions with suggestions for future work.

2 Framework Overview

Instrumentation under SPiKE is facilitated by a technique that we call *Invisi-Drift*. The basic idea involves inserting what we call *drifters*, at memory locations corresponding to the code construct whose instrumentation is desired. Drifters are invisible logical elements that trigger an event when a read, write and/or execute occurs to the corresponding memory location. Each drifter has an associated *instrument*, a user-defined function to which control is transferred, when the drifter triggers. Instruments can then monitor and/or alter the executing code stream as desired. SPiKE's instrumentation strategy is stealth and cannot be detected or countered in any fashion. The framework also employs software techniques such as *redirection* and *localized-executions* to efficiently tackle issues involving reads, writes and/or executes to memory locations with drifters, thereby ensuring instrumentation presence while co-existing with other instrumentation strategies employed by a malware.

SPiKE relies on our stealth breakpoint framework, VAMPiRE (Vasudevan & Yerraballi 2005) to insert drifters at desired memory locations. Figure 1 illustrates the current architecture of SPiKE. The framework core consists of a code slice execute engine (CSXE) and an instrumentation API invoked by the analysis tool that uses SPiKE. The CSXE can be thought of as a pseudo-VM, that enables the framework to execute code fragments (slices) locally. This is needed to efficiently tackle various run-time issues involving SM-SC code (see Section 3.3) and invocation of the original construct at a drifter location (see Section 3.2). The CSXE employs a disassembler and a slice repository (acting as a cache) for its functioning. Since SPiKE's core resides in kernel-mode, it can capture both user- and kernel-mode code with ease.

As Figure 1 shows, there are typically three binary elements present during an analysis session: the target application, the analysis tool front-end, and the tool payloads. The tool payloads contain instruments that need to be in the target address space along with support code that deploys instrumentation. The target address space can be a new process, an existing process and/or the OS kernel itself. The tool front-end runs as a separate process and forms the user interface. The tool front-end initialises the framework and loads the tool payloads in the target address space, by a process that we call payloading (steps 1 and 2). The payloads upon gaining control, use the framework library to initialize SPiKE in the target address space and insert drifters at specified memory locations for instrumentation (step-3). When a drifter triggers (step-4), SPiKE transfers control to the associated instrument, which then does the necessary processing (step-5). Using SPiKE, the tool payloads can communicate with the tool front-end in real time. A point to be noted from Figure 1, is that, payloads in user-mode have a local copy of the CSXE linked to themselves. Thus, localized-executions occur at the same privilege level where the drifter was triggered. This ensures that malformed user-mode code cannot result in system instability, which would have been the case had localized executions always taken place in kernel-mode.

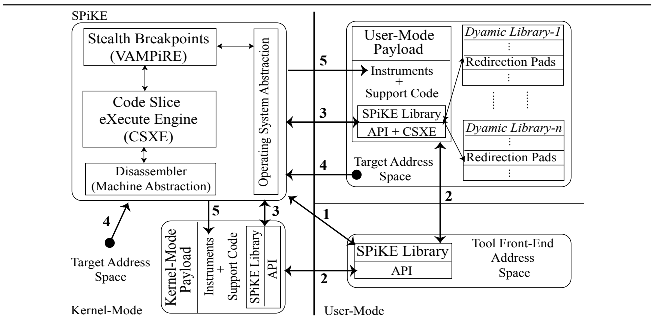


Figure 1: Architecture of SPiKE

SPiKE also employs a technique that we call *redirection*, for efficient instrumentation of functions housed in dynamic libraries. This is achieved by embedding code constructs known as redirection-pads into the target library. SPiKE is completely re-entrant, as it does not make use of any OS specific routines within the CSXE and uses shared memory with its own isolation primitives for interprocess communication. The SPiKE API is simple yet powerful to allow a tool writer to harness the complete power of the framework.

3 Design and Implementation

In this section, we present a detailed dicussion of SPiKE. We begin by discussing how the framework is deployed in the target execution space, followed by a detailed description of how the framework achieves instrumentation. Finally, we discuss the SPiKE API and some issues involving the framework stealthness.

3.1 Payloading

The analysis tool employing SPiKE, uses the framework API, to load the tool-payload(s) into the target execution space. The target execution space can be a new process, an existing process or the OS kernel itself. We call this process *payloading*. The design of our payloading scheme is unified and lends itself to implementation on a variety of OSs such as Windows, Linux, Solaris etc. Payloading is facilitated by the `ptrace` API on Unix systems and process and thread creation APIs such as `CreateProcess`, `OpenProcess`, `Suspend/ResumeThread` etc. on Windows systems. Using our payloading scheme allows us to attach to a new or an already running process in the same way as a debugger. For kernel-mode payloading, a tool-payload simply translates to a kernel-module for the specific OS. For user-mode payloading, a tool-payload translates to a dynamic library. Our payloading mechanism has the ability to track process dependencies (parent and child) and load the tool-payload(s) into newly created children processes automatically, thus aiding local (per process) and global (entire system) instrumentation. Further details regarding the payloading process and issues regarding OSs in the context of payloading can be found in our earlier work, SAKTHI (Vasudevan & Yerraballi 2004).

3.2 Invisi-Drift

SPiKE inserts *drifters* at memory locations corresponding to the code construct whose instrumentation is desired. Drifters are logical elements that are a combination of a *code-breakpoint* and an *instrument.* A code-breakpoint provides the ability to stop execution of code at an arbitrary point during runtime and the instrument is a user-supplied function to which control is transferred when the breakpoint triggers. SPiKE makes use of our stealth breakpoint framework, VAMPiRE (Vasudevan & Yerraballi 2005) to set invisible code-breakpoints at memory locations corresponding to the code constructs whose instrumentation is desired. Stealth breakpoints provide unlimited number of invisible code, data and/or I/O breakpoints that cannot be detected or countered in any fashion. The basic idea involves exploiting the virtual memory system of the underlying platform by setting the attribute of the target memory page containing the memory location to *not-present* and using the page-fault exception (PFE) and subtle software techniques to trigger the corresponding breakpoint (Vasudevan & Yerraballi 2005).

A drifter under SPiKE can have a local (per-process) or global (entire system) presence and can be active or inactive at a given instance. A drifter can trigger (due to the triggering of the corresponding code-breakpoint) due to a read, write and/or execute to the memory location where it is inserted. When a drifter triggers, SPiKE invokes the instrument corresponding to the drifter, which can then monitor and/or alter program behavior by overwriting registers and/or memory (including the stack). They can also invoke the original code at the drifter location within themselves. This feature could be used to chain to the original construct to do the meat of the processing while employing changes to the result. As an example, let us consider the Windows OS kernel API `CreateProcess`. This API is responsible for creating new processes in the system. A possible instrumentation of this API might be used to keep track of the processes that are being created in the system and their relationships (parent/children). Under SPiKE, the instrument for the `CreateProcess` API can invoke the original API to create a process, perform its function (which is to keep track of the process hierarchy) and return to the caller. This model is much more flexible than a function prologue/epilogue instrumentation as found in probe-based systems such as DynInst (Buck & Hollingsworth 2000), DProbes (Moore 2001) etc. in that: (a) it decouples the instrument from the construct being instrumented, which enables the instrument to invoke the original construct as many times as required with different sets of parameters on the call stack and (b) it is a generic method that can capture invocations to constructs that are not pure. Instruments also have the ability to insert and/or remove drifters at any memory location dynamically.

3.3 Localized-Executions

With Invisi-Drift, when a drifter is inserted at a desired memory location, the entire memory page corresponding to the location, has its attribute set to *not-present* as described in the previous section. Further, for instrumentation persistence, such memory pages always need their attribute set to *not-present* for the lifetime of a drifter. These facts give rise to a couple of issues that need to be dealt with. The first is when the target code reads and/or writes to drifter locations or to locations within the memory page containing drifters. The second situation is when the original code at the drifter location is invoked from within the instrument. Both cases result in multiple PFEs due to the destination memory page attribute being *not-present*.

The framework employs the CSXE to eliminate such PFEs and execute such code with minimal latency. The CSXE *slices* code at a given location and stores it in a *slice-repository.* A slice is nothing but a straight line sequence of instructions that terminates in either of these conditions: (1) an unconditional control transfer, (2) a specified number of conditional control transfers or (3) a specified number of instructions. The slice-repository acts as a framework local cache and contains the collection of code sequences that have been sliced. Only code residing in the slice-repository is executed-never the original code and hence the term *localized-executions.*

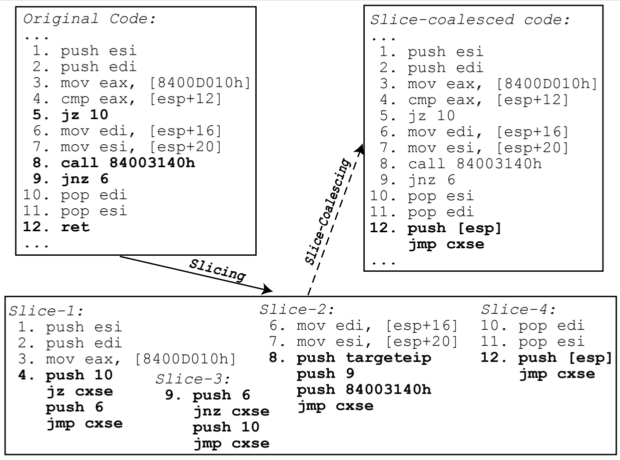


Figure 2: Localized-Executions using CSXE

Figure 3 illustrates a typical slicing process. A slice consists of one or more exit sequences which transfer control to the CSXE. The CSXE determines the target address, performs slicing for the target if it has not done before, and resumes execution at the target slice. Branches to and from the CSXE result in the saving of the current registers and flags. Generating exit stubs for unconditional control transfers is as simple as performing an unconditional jump into the CSXE. For certain unconditionals such as the `call` instruction, the stub pushes the destination instruction pointer (targeteip, line 8, slice-2, figure 2) and performs an unconditional jump to the CSXE. This ensures that code employing position independent access (as seen in many SM-SC code) works without any problems. For unconditional brances, the CSXE employs a host of exit stubs for different variants (slice-1 and slice-4, figure 2). The basic idea being to translate a conditional into a conditional and an explicit branch. This model enables *slice-coalescing*, whereby a group of slices can be coalesced to minimize exits to the CSXE. Slice-coalescing is a powerful mechanism that can produce code fragments which are very similar to the original code that can be locally executed with minimal latency. The CSXE also stores what are known as *ghosts* for read and/or writes to memory pages containing drifters. These are duplicates of the original memory page where drifters are inserted and are used when the original code at the drifter location need to be invoked within the instrument. Ghosts allow others to install their own instruments while preserving the original function for invocation by the framework instruments.

The CSXE differs from traditional VMs employed in current JIT schemes in that it does not reallocate registers or perform any liveliness analysis etc. This eliminates the latency associated with dynamic compilation since localized execution is typically over a limited code fragment (maximum of a page size). The CSXE stops slicing if the target address of goes beyond the memory page containing the drifter for instances where the code residing at the drifter location needs to be invoked from the instrument. For situations involving reads and/or writes to drifter locations or locations within the memory page of the drifter, the CSXE stops code slicing after a certain number of instructions that are obtained and refined over time by employing simple heuristics on the target code. Eg. A sequence of writes to a given location, often happen with a loop over a specified number of instructions. The fact that the CSXE is completely re-entrant, does not re-allocate registers, does not tamper with the code stack and employs subtle exit stubs for conditional and unconditionals, endows it with the capability to support multithreading and SM-SC code — features that are unavailable in current JIT schemes.

3.4 Redirection

SPiKE allows drifters to be inserted at arbitrary memory locations for instrumentation purposes. However, this direct method of instrumentation incurs a latency when it comes to invoking the code at the drifter location from within the instrument due to localized-executions as described in the previous section. This latency can be nullified if we hinge on the fact that most of the services offered by the OS or an application are in the form of dynamic link libraries (DLL) or shared libraries. For example, the socket `send` function is provided as an export of the DLL, `WINSOCK.DLL` under the Windows OSs and as an export of the shared library, `libsocket.so` under the Linux OS. While there are programs that make use of static libraries, their numbers are very limited (about 1% of all applications in our experiment). Also on some OSs such as Windows it is simply not possible to statically link against the system libraries without severely compromising compatibility.

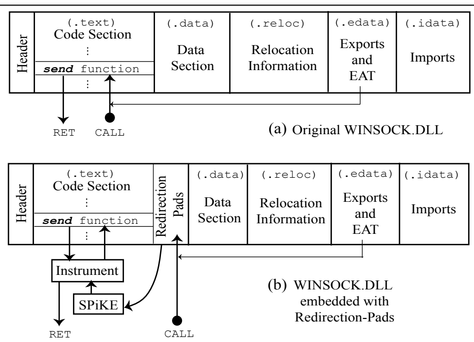


Figure 3: Redirection for Dynamic Libraries

Every DLL or a shared library exports the functions contained within it using an *export-table*. Among other fields, the export-table contains an array of pointers to functions within the library known as the *Export Address Table* (EAT). When an application links with a DLL or a shared library, it *imports* a set of functions from it. This is in the form of an *Import Address table* (IAT) or a *Procedure Linkage Table* (PLT) within the application. Further details regarding the executable formats of DLLs and shared libraries can be found in the Portable Executable Specification (Microsoft Corporation 2004) and the Executable and Link Format Specification (Unix System Laboratories 1998). The OS loader, obtains a list of the functions that are being imported, loads the appropriate libraries, fetches the addresses of the functions using the EAT and patches the IAT or the PLT within the application. Figure 3a shows a sample function invocation from an application using the DLL, `WINSOCK.DLL`.

SPiKE re-writes the DLL or the shared library that contains the function(s) to be instrumented, with redirection pads. The re-writing is carried out in binary and does not require the sources and/or any debug symbols for the library. Every function in the module, that is intended to be instrumented will have a unique *redirection-pad* embedded within the module. A redirection-pad is a sequence of instructions which finally transfer control to the function that is associated with it. The export-table entries of the module are then overwritten to point to the redirection pads of the corresponding functions. Thus, when an application links to the library and imports a function which has a redirection-pad, it will in reality invoke the redirection-pad code instead of the original function, when the target library is loaded and the OS loader performs the IAT patch. However, the semantics of the function invocation will remain the same as the redirection-pads chain to the original function without any changes. Figure 3b shows the same function invocation for the DLL, `WINSOCK.DLL` with the redirection-pads in place. Note from the figure that the redirection-pads are embedded directly in the code section of the library and not as a separate section. This is needed to hide the framework from the malware being analyzed (see Section 3.6).

Thus, when drifters are inserted at memory locations corresponding to the functions within the DLL or shared libraries, they will be inserted on memory pages containing the redirection pads. In other words, the original function is available for invocation from within the instrument with no latency involved. The redirection-pads however, still incur a latency due to reads and/or writes but as a side-effect elminate the need for ghost pages, since any modifications to the original code is in fact modifying the redirection-pad code. The instructions generated in the redirection-pads are polymorphic to prevent the malware from detecting the framework using pattern-based schemes on the redirection-pads (see Section 3.6). The framework also allows the user to tune the number of redirection-pads per memory page, and the range for the number of instructions that will be generated for a redirection-pad.

3.5 SPiKE API

Anaylsis tools that employ SPiKE are usually written in C/C++ using SPiKE's API. The API is easy-to-use and is designed to be platform independent whenever possible allowing tool code to be re-usable while allowing them to acess platform specific details when necessary. In Figure 4, we list a partial code of one our preliminary tool employing SPiKE under the Windows OS. The tool, `DrvDllMon`, enables complete monitoring of drivers and dynamic link libraries that are being loaded and unloaded in the system.

The main interface to SPiKE is provided in the form of five easy to use API functions, `spike_init`, `spike_payload`, `spike_insertdrifter`, `spike_removedrifter` and `spike_comm`. As seen from Figure 4, a tool using SPiKE is typically composed of two elements. A tool payload and a tool front-end. The tool payload contains the instruments that need to be in the target address space along with support code that deploys instrumentation. The tool front-end forms the user interface. The front-end of the tool initializes the framework and typically invokes the `spike_payload` API to load the payload into the target address space. The `spike_init` API is used to initialize the framework. The payload upon gaining control, initializes the framework in the target address space, and uses the `spike_insertdrifter` and `spike_removedrifter` API calls to insert or remove drifters at selected memory locations. As mentioned earlier, drifters can have a global (`DGLOB`) or local presence (`DLOCL`) with the ability to trigger on a read (`DREAD`), write (`DWRITE`) and/or execute (`DEXEC`) to its corresponding memory location. Communication between the payload and the tool front-end occurs via SPiKEs API `spike_comm`.

Note that the payload code is automatically translated into the appropriate executable during compile time, depending on whether instrumentation is desired in user- and/or kernel-mode. The instruments access relevant information via the argument of type `DRIFTERINFO`, which among other fields includes a pointer to the current stack and a function pointer that can be used to invoke the original construct at the drifter location from the instrument. The instruments can thus alter program registers, memory (including the stack) and perform any semantic changes. The instruments can also access instrument specific data using the `exinfo` field of the argument. This feature, for example allows a single instrument to be associated with multiple drifters as seen from Figure 4 where the instrument `LL_Inst` is associated with drifters to the ANSI (`LoadLibraryA`) and UNICODE (`LoadLibraryW`) versions of the API `LoadLibrary`, which is used to load a DLL under the Windows OS. SPiKE also provides a host of other support APIs apart from the ones discussed above, related to architecture and OS specific elements such as instructions, processes, libraries, files etc.

```
---drvdllmonp.c (drvdllmon payload)---
#include <spike.h>
...
void LL_Inst(DRIFTERINFO *di){ \\LoadLibrary API Instrument
 ...
 if(di->exinfo[0] == 'W') \\UNICODE version
  spike_comm("0x%X: LoadLibW(%ls)", di->stack[0], di->stack[4]);
  ...
 di->retval= di->originalcode(di->stack[4], di->stack[8]);
}

void payload_init(PROCESSINFO *p){ \\payload initialise function
 ...
 spike_init(); \\initialise framework in process address space
 ...
 \\instrument the ANSI and UNICODE version of Loadlibrary API
 addr1=spike_addr("kernel32.dll", "LoadLibraryA");
 addr2=spike_addr("kernel32.dll", "LoadLibraryW");
 spike_insertdrifter(addr1, DEXEC | DGLOB, LL_Inst, 'A');
 spike_insertdrifter(addr2, DEXEC | DGLOB, LL_Inst, 'W');
 ...
}
...

---drvdllmonf.c (drvdllmon front-end)---
#include <spike.h>
...
int main(){
 ...
 spike_init(); \\initialise framework
 ...
 spike_payload(pid, "drvdllmonp"); \\load the payload
 ...
}
...
```

Figure 4: DrvDllMon, a tool employing SPiKE

3.6 Stealth Techniques

SPiKE uses drifters, CSXE and redirection-pads for implementing un-obtrusive instrumentation. However, these elements and the latency that they introduce can be detected during runtime albeit with some subtle tricks. We now present some simple techniques that the framework employs for stealthness to counter such methods. While there are a host of detection methods and their antidotes, due to space constriants, we will only concentrate on a few important ones.

Drifters rely on the virtual memory system for their operation. However, drifter triggerings result in increased latency due to PFEs being invoked that is not present during normal execution. A malware could use this fact to detect if its being analyzed. As an example, on the IA-32, a malware could use the `RDTSC` instruction (returns the number of clock cycles that have elapsed since system-bootup) and/or the real-time clock to obtain a relative measure of its code fragment execution time. Depending on the system SPiKE is run under, the framework applies a clock patch resetting the time stamp counter and/or the real-time clock to mimic a value close to that of a normal execution.

SPiKE makes use of redirection-pads within the dynamic libraries to void latency issues during invocation of original constructs from the instruments. The redirection pads are embedded within the library in the same section as the function code. This is necessary since, if the pads were embedded in a separate section, a malware could check the EAT entries and compare them with the library header to detect that the EAT entries have been routed. However, since the redirection-pads are within the code section of the library, there is no way of distinguishing them from the library functions. The redirection pad code is polymorphic in the sense that one cannot employ any signature detection to check for them. The framework employs a polymorphic code generator, that allows fine tuning of the upper-bound of the number of instructions that will be generated in a redirection-pad. In other words, a malware cannot search for a unique instruction within the redirection pad code which would establish a postive detection of the framework.

SPiKE employs the CSXE for localized code execution during read, writes or executes to drifter locations. Though the CSxE supports SM-SC code, a malware could use a technique such as to single-step its own instruction stream and performing the actual operation within its single-step handler. The malware single-step handler can check the return address on the exception stack and figure that the code is not being executed at the address it should have been (since localized-executions execute the slice from a different memory location altogether). The CSXE can recognize such situations, and will execute the code slice instructions one by one invoking the target single-step handler with the original instruction addresses.

The framework also employs a polymorphic engine to ensure that every instance of its deployment is different in the form of any privileged modules, environment variables, configuration files and code streams. Thus, no malware can detect SPiKE by searching these elements for a pattern.

4 Experience

In this section we discuss our experience with one of our tools employing SPiKE to analyze a real world malware, therby illustrating the framework utility. Our tool called `WatchDog`, currently runs under the Windows OS, and enables monitoring of various user- and kernel-mode components of the OS, offering insight into malware behavior with respect to the system. `WatchDog` is a simple yet versatile tool that allows real-time monitoring of Windows APIs related to files, processes, registry, network, memory and various other sections of the OS both in user- and kernel-mode. It enables complete monitoring of drivers and dynamic link libraries that are being loaded and unloaded in the system. The tool also allows instrumenting the entry and exit points of the dynamic link libraries and drivers with support for EAT access monitoring, critical data access monitoring, dependencies, per process filters and a host of other features which make it an indispensable tool to monitor malware behavior in a system. `WatchDog` is one of the many components that make up our malware analysis environment that is currently under research and development. This section will discuss our experience using `WatchDog` on several monitoring sessions with a Windows based trojan, `W32.MyDoom`. We chose `W32.MyDoom` as our candidate for discussion, since it has several variants, employing a variety of anti-analysis tricks that one would typically encounter in recent malware.

The `W32.MyDoom`, with variants commonly known as `W32.MyDoom.X-mm` (where X can be S, G, M etc.) is a multistage malware. It generally arrives via an executable e-mail. Once infected, the malware will gather e-mail addresses, and send out copies of itself using its own SMTP engine. It also downloads and installs backdoors and listens for commands. Once installed, the backdoor may disable antivirus and security software, and other services. The downloaded trojan might allow external users to relay mail through random ports, and to use the victim's machine as an HTTP proxy. The trojans downloaded by the `W32.MyDoom` envelope, generally posses the ability to uninstall or update themselves, and to download files.

The `W32.MyDoom` and modified strains cannot be completely analysed using traditional JIT and/or probe-based binary-instrumentation. The malware employs a polymorphic code envelope and employs efficient anti-probe techniques to detect probes and will remain dormant and/or create system instability in such cases. For the purposes of discussion, we look at a simplified code fragments of different variants of the `W32.MyDoom` under monitoring sessions with `WatchDog` and our malware analysis environment. The fragments are shown in the

IA-32 assembly language syntax. We have removed details from the code that are not pertinent to our discussion and have restructured the code for easy understanding. Consider a fragment of code shown in Figure 5a.

```
    ...
 1. mov eax, [esi+3ch]
 2. mov ecx, [esi+3fh]
 3. mov edx, [esi+4bh]
 4. mov ebx, [edx]
 5. cmp ebx, [esi+3ah]
 6. jb BADBEHAVE
 7. cmp ebx, eax
 8. ja BADBEHAVE
 9. loop 4
10. mov edi, [esi+2ah]
11. call 14
12. jnz BADBEHAVE
13. call install_hooks
    ...
```

```
    ...
    ;procedure check_hooks
14. push edi
15. push esi
16. call get_instruction
17. cmp byte ptr [edx], 0E9h
18. je check_jumps
19. cmp byte ptr [edx], 0CDh
20. je check_traps
21. cmp byte ptr [edx], 0F0h
22. je check_invalids
    ...
23. ret
    ...
```

(a) Instrumentation Check Code Fragment

```
......
CS:112050DCh CreateFile("%SYSTEMROOT%\WINVPN32.EXE", ...) = F124h
......
CS:0042312Ah winsock.dll!send(1A20h, [DS:00404050h], ...) = 800h
CS:11204A2Bh a32ss32.dll!send(1A20h, [DS:00404050h], ...) = 800h
CS:112051E0h a32ss32.dll!connect(213Bh, 84.168.21.11, 200) = 0h
......
CS:0044A340h winsock.dll!send(1A20h, [DS:00404010h], ...) = 400h
CS:11204A2Bh a32ss32.dll!send(1A20h, [DS:00404010h], ...) = 400h
CS:11205AE4h a32ss32.dll!send(213Bh, [DS:1121A000h], ...) = C00h
CS:11206120h a32ss32.dll!recv(213Bh, [DS:12005000h], ...) = 200h
CS:11206120h a32ss32.dll!recv(213Bh, [DS:12102000h], ...) = 200h
......
CS:1120AE70h WriteFile(F124h, [DS:12005000h], ...) = 1000h
CS:1120AE70h WriteFile(F124h, [DS:12102000h], ...) = 1000h
......
```

(b) Behavior Log showing a Trojan Download

```
    ...
 1. mov eax, [edx+1b]
 2. mov edi, [edx+2a]
 3. mov ecx, [edx+5b]
 4. mov esi, [edi]
 5. cmp esi, eax
 6. jb BADBEHAVE
 7. cmp esi, [edx+10]
 8. ja BADBEHAVE
 9. loop 4
    ...
10. call 12
    ...
11. call ebx
    ...
```

```
    ...
    ;procedure check_function
12. mov eax, [edx+10]
13. mov esi, [edx+56]
14. mov ecx, [edx+10];
15. add ebx, [esi]
16. rol ebx, 8
17. loop 15
    ...
18. ret ;ebx=address of virus
           internal function
    ...
```

(c) Integrity Check Code Fragment

Figure 5: (a)-(c) W32.MyDoom Anti-analysis Tricks and Behavior Log

The W32.MyDoom and its variants instrument (hook) several APIs within the system. One such API that is hooked is the ZwQueryDirectoryFile. The ZwQueryDirectoryFile API under the Windows OSs (NT, 2K and XP), is a kernel function that is used to obtain the list of files within a particular directory. Calls to upper level file functions are ultimately translated to invoke this API within the kernel. The malware hooks this API, so that it modifies the return query buffer in a way that would prevent any regular application from seeing its critical files. There are several ways to instrument this API. The first method is by changing the pointers in the system service table pointed to by KiSystemServiceDispatcherTable to point to the instrumented functions. The pointer location for a particular system service can be found out by disassembling the dynamic library, NTDLL.DLL to get the indices for several system-calls. The second method is by using a probe-based instrumentation on the starting address of the API. Thus, if one instruments the API before-hand, using either of the two methods, it is easy to observe the behaviour of the malware, even if it hooks the API once again (since the malware will have to invoke the original API to obtain the populated query buffer in the first place).

However, the W32.MyDoom is intelligent to spot this. It performs a couple of checks to see if the API has been already hooked. The first check is a detection of a system-table hook (lines 1-9, Figure 5a). The malware uses a fact that original system-table entries point to code within NTOSKRNL.EXE, the Windows OS kernel. Thus, if one were to change the system-table entries to point to their own code, it will be at an address that is outside the image-space of NTOSKRNL.EXE. If this detection passes, the malware jumps to a location within its code which leads to a dormant operation or in certain cases causes system instability.

The second form of detection is for probe-based schemes at the target API. All probe-based schemes rely on the use of a unconditional transfer instruction (branch, trap etc.) at the address of the target code to be instrumented. This makes them an easy prey to detection. As seen from lines 14-22 of procedure check_hooks of Figure 5a, the malware checks to see if the code at the target address is any instruction that could potentially transfer control to an instrument (jump, trap, invalid instructions etc.). The check sweeps through the target API address checking for such instructions before a conditional is encountered. Since current probe-based frameworks rely on the use of control transfer instructions at the start of the target function thats being instrumented, they are rendered unusable in this context. The idea of inserting the probe, after the malware has inserted its hook doesnt accomplish any functionality since it would never see the unmodified buffer, but is also defeated since there are several checks, scattered throughout the code of W32.MyDoom, within its polymorphic layers, to ensure that no other system can hook the APIs after it has.

However, using SPiKE it is trivial to instrument such APIs, while at the same time allowing the malware hook to be active. In effect, WatchDog bypasses the checks and can monitor such APIs in a stealth fashion. Figure 5b shows a sample log from the utility revealing a trojan download and implant.

Certain variants of the W32.MyDoom use localized DLL's for their operation. For example, the trojan renames WINSOCK.DLL, the dynamic library associated with socket functions, to a random name and replaces it with its own DLL instead. The replaced DLL is coded in a fashion employing polymorphic layers and incorporates malware functionality while finally chaining to the functions within the renamed WINSOCK.DLL. As seen from the log of Figure 5b, WatchDog captures both the replaced (WINSOCK.DLL) and the renamed (A32SS32.DLL) dynamic library invocations, allowing us to spot an activity such as a trojan download and implantation as shown. The tool enables logging of several important pieces of a function invocation such as the parameters, the return value, the location it was invoked from, the stack and memory contents etc. WatchDog also features what we call *selective logging*, whereby only invocations arising from a specified range of memory is monitored. This is a very useful technique that helps maintain the clarity of information that is logged. The tool also features a script based interface allowing users to add real-time actions to monitored events.

The localized DLL's of W32.MyDoom employ certain anti-probing tricks. We noticed a couple of such tricks in one variant during our analysis. The code fragment of the malware in this context is shown in Figure 5c. The malware employs integrity checks using checksums on the socket primitives on the replaced WINSOCK.DLL (check_function procedure, Figure 5c). Thus, inserting traditional probes on such functions, results in erratic behavior (lines 6 and 8, Figure 5c). Solutions employing replacing the replaced DLL or rehousing the EAT entries in the replaced WINSOCK.DLL are defeated by similar checksum fragments that are embedded within the malware code. If the malware detects a malformed replaced DLL, it will overwrite it with a new copy. As an added detection, the malware also checks for probe insertions in the renamed DLL (A32SS32.DLL) once loaded. Manual patching of such integrity checks are cumbersome since: (a) many such fragments are scattered throughout the malware code and (b) the checksum themselves are used as representatives of the target address of a call that performs some processing pertaining to the malware functionality (line 18 and 11, Figure 5c). Thus, on a valid checksum the call performs the desired internal function whereas on an incorrect checksum, it branches to a location where the code is nothing but garbage. With SPiKE, instrumenting functions within the renamed DLL is trivial using redirection-pads. For instrumentation of the replaced DLL, drifters are inserted at desired functions directly.

As seen, traditional probe-based insturmentation methods do not suffice to study code employing self-modification, self-checking, hooking and/or any form of anti-analysis and/or anti-debugging schemes as in the case of the `W32.MyDoom` and other similar malware. With SPiKE however, this task is greatly simplified. The framework allows instrumentation of the target code without any form of code modification in a stealth fashion making it very hard to detect and counter. The latency of the framework is well suited for interactive monitoring (as seen from its performance measurements in the next section). These features combined with the fact that the recent trend in malware has been to employ schemes to detect and prevent any form of analysis, make SPiKE the first and a very powerful instrumentation framework to aid in the construction of malware analysis tools.

5 Performance Evaluation

In this section, we first report the performance of SPiKE without any active instrumentation. We then report the performance of SPiKE's direct (without redirection) and redirection-based instrumentation methods. Finally, we compare SPiKE with some widely used JIT and probe-based tools, and show that the framework incurs low latencies while providing features that are highly conducive to malware analysis.

Before we proceed to discuss the performance measurements of the framework, a few words regarding the test-bench are in order. The current version of SPiKE supports IA-32 (and compatible) processors running the Windows and Linux OSs. Our experimental setup consists of, an AMD Athlon XP 1.8 GHz processor with 512MB of RAM, running Windows XP and Windows 2000, and an Intel Xeon 1.7 Ghz processor with 512MB of RAM running Linux. To validate SPiKE, we use our analysis tool `WatchDog` (see Section 4). We use processor clock cycles as the performance metric for our measurements. This metric is chosen, as it does not vary across processor speeds and also since it is a standard in literature related to micro-benchmarks. The `RDTSC` instruction is used to measure the clock cycles. Unless otherwise specified, for the graphs encountered in the following sub-sections, the x-axis is the amount of extra clock cycles that are incurred as opposed to the native run-time of the function and the latency is measured by executing sufficient runs and averaging the values to obtain a confidence interval of 95%. Also, parts of some graphs are magnified (indicated by dotted lines) to provide a clear representation of categories with low values on the x-axis.

5.1 Latency Without Instrumentation

SPiKE has zero instrumentation effect for each instrumentation point for inactive direct instrumentation. This is because, a drifter inserted at a memory location can only trigger when the corresponding memory page attribute is set to *not-present* upon drifter activation. However, SPiKE's redirection-based instrumentation incurs a non-zero, but low run-time overhead when instrumentation is inactive. This is due to the execution of the instructions in the redirection-pads corresponding to the functions to be instrumented. The number of these instructions are variable and depend upon a user-defined range for a session (see Section 3.4). The framework latency for various dynamic library functions, from different analysis sessions under the Windows OSs, for three arbitrary user-defined ranges are as shown in Figure 6a.

The average latency for the functions when using the user-defined ranges shown in Figure 6a, are 343, 1405 and 2910 clock cycles respectively. For a 1.8 GHz processor, these numbers average to $.19\mu s$, $0.78\mu s$ and $1.6\mu s$ respectively, which is minimal. In general, the more the number of instructions per redirection-pad, the more is the latency and the less is the chance of its detection. In our opinion a user-defined range of 15–50 instructions should suffice to keep the redirection-pads from being detected using any patterns. A point to be noted from the graph is that, for the same function (`NtReadFile(1)` in Figure 6a for example), one could have different latencies per analysis session for the same user-defined range. This is because of the polymorphic nature of code that is generated in its redirection-pad.

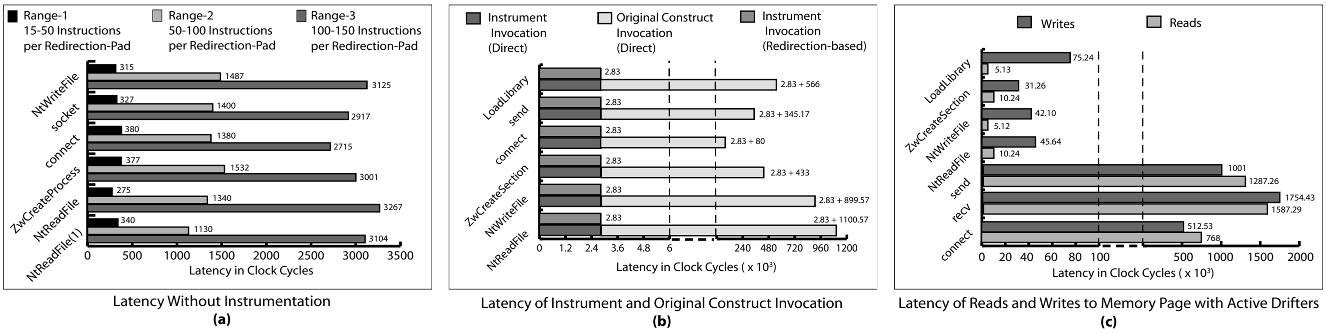
5.2 Latency With Instrumentation

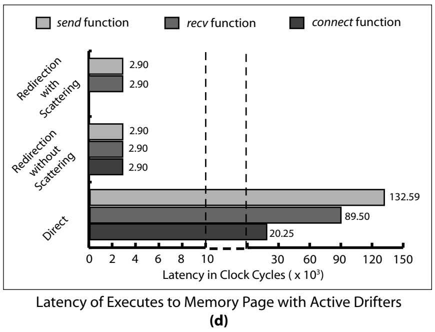
We now study the performance of SPiKE with active direct and redirection-based instrumentation. We divide the total run-time overhead into three components: (a) latency due to instrument invocation, (b) latency due to invocation of the original construct at a drifter location from the instrument, and (c) latency due to reads, writes and/or executes to a memory page containing active drifters.

The instrument invocation time is the time that has elapsed after the transfer of control to a code construct (control transfer is usually done via a `call` for a code construct that is a function), and before control is handed over to the instrument. Figure 6b shows the latency involved in invoking the instrument and the original construct at a drifter location, for both direct and redirection-based instrumentation of SPiKE, for various arbitrary functions within DLLs and shared libraries under the Windows and Linux OSs. As seen from Figure 6b, both direct and redirection-based methods incur the same instrument invocation latency. This is due drifter triggering, which incurs a constant processing overhead for any instrumented memory location due to the invocation of the PFE (see Section 3.2). However, considering the latency involved in invoking the original construct at a drifter location, the direct method incurs a higher performance penalty when compared to the redirection-based method. This is due to the framework employing the CSXE for localized-executions in the direct method. In contrast, the redirection-based method uses redirection-pads to eliminate this overhead.

The framework also incurs other forms of latency in the form of reads, writes and/or executes to a memory page containing active drifters. For example, a malware might try to install its own instrumentation at a memory location where a drifter is inserted. This would result in PFEs due to reads and/or writes to the drifter location. Another example would be when a drifter is inserted at the start of a function, but the memory page containing the function also houses parts of another function that is not instrumented. Thus, executes to the uninstrumented function also trigger PFEs. SPiKE employs the CSXE to tackle such cases which results in the aforementioned latency. Coming up with representative inputs for such cases, for purposes of performance evaluation, is not an easy task since they depend on a lot of factors, chief among them being the method of analysis adopted by an individual, the structure of the executing code and dependency of one function over the other, which are not easily characterized. Thus, we will concentrate on presenting the performance of the framework related to some specific code fragments for such cases, and show that the framework latency is within limits to suit interactive analysis. The performance of the framework for other situations can be estimated in a similar fashion.

Consider the code fragment in Figure 5a. This fragment of the `W32.MyDoom` trojan checks to see if a certain group of system functions have been instrumented prior to its execution (see Section 4). As seen from Figure 5a, the procedure `check_hooks` reads every instruction from the memory locations corresponding to the func-





Feature / Framework	Transparent	Original Construct Invocation	Success Rate	Kernel-Mode Support	Stealth	Multithreading	Arbitrary Code	SM/SC Code	Platform Support
DProbes	✗	✗	< 100%	✓	✗	✓	✗	✗	Linux/IA-32, IA-64
Detours	✗	✓	< 100%	✓	✗	✓	✗	✗	Windows/ IA-32
DynInst	✗	✗	< 100%	✓	✗	✓	✗	✗	Windows, Linux/ IA-32
Pin	✓	✗	100%	✗	✗	✗	✓	✗	Linux/ IA-32, IA-64 EMT64, ARM
DynamoRIO	✓	✗	100%	✗	✗	✗	✓	✗	Windows, Linux/ IA-32
SPiKE	✓	✓	100%	✓	✓	✓	✓	✓	Windows, Linux/ IA-32

Feature Comparison
(e)

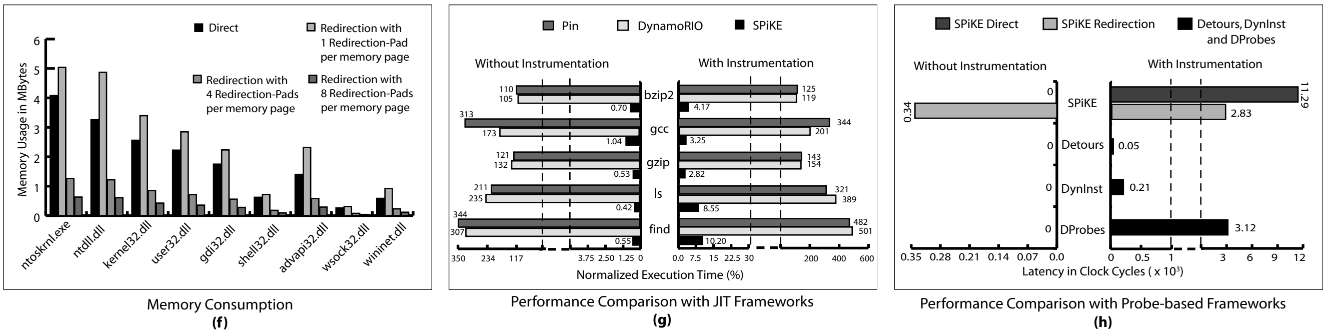


Figure 6: (a)-(h) SPiKE's Quantitative and Qualitative Evaluations

tions, checking for inserted probes. If this check passes, the virus uses the procedure `install_hooks` to write out its own probes at desired memory locations. Figure 6c shows the latency incurred by the framework due to such reads and writes on certain system functions checked and instrumented by the procedures `check_hooks` and `install_hooks` and to functions housed in localized DLLs employed by the virus (in this case `WINSOCK.DLL`), using both direct and redirection based instrumentation. One can observe from the graph that for functions housed in system and/or standard libraries, the number of reads and/or writes is very minimal (mostly the first few instruction of the function), whereas for a malware specific DLL that overwrites its own code, the latency is higher on account of multiple reads and/or writes due to its polymorphic nature. In either cases, the latency is well within the limits to suit interactive monitoring and analysis.

Now, let us consider the behavior log shown in Figure 5b, that reveals a trojan download (see Section 4). Let us consider the APIs `send`, `recv` and `connect` of the dynamic library `A32SS32.DLL` and assume that only `send` and `recv` are instrumented using SPiKE. Figure 6d shows the latency incurred (instrument invocation and original construct invocation put together) for both direct and redirection-based schemes of instrumentation, for a single invocation of the three APIs. As seen, in the direct method, the function `connect` incurs an overhead even though it is not involved in instrumentation. This is due to the fact that APIs `send` and `connect` share the same memory page in `AS23SS32.DLL`. Thus, the framework needs to employ localized-executions using the CSXE for the `connect` function which explains the latency.

However, when it comes to the redirection-based method, the latency of the framework depends upon the number of redirectors per memory page and the way the redirection pads are chosen for the functions. As an example consider the graph in Figure 6d which shows the latency of the three APIs, with 4 redirection pads per memory page, and assigned redirection-pads that fall on the same memory page. This results in a higher runtime latency, since APIs that are not involved in instrumentation, but lie within the same memory-page incur the overhead of localized-executions. However, by carefully assigning redirection-pads for the functions such that they do not fall on the same memory page, the latency can be reduced. This is shown in Figure 6d, with 4 redirectors per memory page, but with the APIs assigned redirection pads in such a fashion so that they do not occupy the same memory page. This technique, that we call *scattering*, can be applied to functions housed in DLLs and/or shared libraries and helps in reducing the latency involved in executing functions that are not involved in instrumentation but share the same memory page with functions that are instrumented.

5.3 Memory Consumption

The direct and redirection based instrumentation strategies of SPiKE incur a memory overhead due to their design elements. The direct method has zero memory overhead when instrumentation is inactive. For active instrumentation using the direct method, every memory page containing a drifter incurs a memory overhead of one extra memory page. This extra memory page, called a *ghost*, is used by the framework CSXE for localized-executions (see Section 3.3). The redirection-based method on the other hand incurs a memory overhead when instrumentation is both active or inactive. This is due to the redirection-pads being embedded in the DLLs and/or shared libraries which consume memory irrespective of any instrumentation.

The exact memory overhead for the direct method, depends on the number of drifters and the memory locations at which they are inserted. Similarly, for the redirection-based method, the actual memory overhead depends on the number of redirection-pads per memory page and the number of functions that are instrumented in a DLL and/or shared library. These factors depend on the nature of analysis employed by an individual which as mentioned before is not easy to characterize. Thus, we will concentrate on presenting the memory overhead of the framework in the context of a specific analysis session under the Windows OS. The memory overhead of the framework for other situations can be estimated in a similar fashion.

Figure 6f shows the memory overhead for both direct and redirection-based instrumentation methods of SPiKE, for selected dynamic libraries, from a session used to analyse the `W32.MyDoom` trojan under the Windows OS (see Section 4). Readings were obtained by instrumenting all the exported functions within each DLL and/or shared library using the direct method as well as the redirection-based method with arbitrary number of redirection pads per memory page. As seen from Figure 6f, the worst-case memory consumption (all functions instrumented within all selected libraries with 1 redirection-pad per memory page) is around 23MB which is not very demanding. A point to be noted from the graph in Figure 6f is that, the higher the number of redirection-pads per memory page, the lesser is the memory consumed. However, choosing too high a value might have effects on techniques such as *scattering* and the framework stealthness. In our opinion, a value of 4 to 16 for the number of redirection-pads per memory page suffices to keep the framework memory utilization minimal while ensuring its efficacy.

5.4 Framework Comparison

We now compare SPiKE against some popular JIT and probe-based instrumentation frameworks such as Pin, DynamoRIO, Detours, Dyninst and DProbes on the IA-32 (and compatible) processors. Detours runs exclusively on the Windows OS, DProbes and Pin run exclusively on the Linux OS, while DynamoRIO and DynInst run on both OSs. We used the latest release of each framework for this experiment: Pin Kit 2411 (Luk et al. 2005), DynamoRIO 0.9.4 (Bruening 2004), Detours 2.0 (Hunt & Brubacher 1999), Dyninst 4.2.1 (Buck & Hollingsworth 2000) and DProbes 2.6.9 (Moore 2001). The first part of this section discusses the features provided by these JIT and probe-based instrumentation frameworks and how SPiKE compares qualitatively. A quantitative comparison in the later part sheds light on the overhead involved in applying SPiKE's instrumentation when compared to the other frameworks.

Figure 6e shows various attributes of an instrumentation framework and how SPiKE compares qualitatively to various JIT and probe-based frameworks. As seen, SPiKE though standing out in features supported in the context of malware (transparent, stealth, multithreading, SM-SC code and arbitrary code), also offers general capabilties that match (in certain cases better, such as original construct invocation, kernel-mode support and success-rate) that of the existing frameworks.

For a quantitative comparision, coming up with a representative performance evaluation criteria was difficult since not all the features offered by SPiKE is available on other frameworks. For example, no other existing framework except for Detours offers the ability to invoke the original construct at the instrumented location. JIT frameworks do not support instrumentation in kernel-mode and only a few of them allow instrumentation to be set at a function level in a clean fashion. (e.g. Pin does, but DynamoRIO does not). Also as mentioned before not all the JIT and probe-based frameworks are supported under various OSs. Thus, for our comparision we chose to measure only the intrument invocation time under the Linux OS for the frameworks (except for Detours). For Detours, we chose to compare both the instrument invocation time and the time for invoking the original construct at the instrumented location under the Windows OS. Considering that SPiKE scores over all the frameworks in terms of the features provided and the fact that our main aim is to show that the SPiKE's instrumentation achieves a low latency, the performance criteria we have chosen is acceptable.

Figure 6g shows the performance of SPiKE when compared to JIT frameworks. Since JIT frameworks are VM based, their instrument invocation time depends on the nature of the executing code. Hence, for our comparision we instrumented the file `stat` API for various applications under Linux and measured the normalized execution time of the applications both with and without instrumentation. The instrument for the file `stat` API had no processing whatsoever. This allowed us to measure and compare the instrument invocation time. As seen from the figure, SPiKEs instrumentation overhead is very minimal when compared to that of the JIT frameworks both with and without instrumentation.

Figure 6h shows the performance of SPiKE when compared to probe-based frameworks. The instrumentation latency of probe-based frameworks are not dependent on the nature of code that is executed. Hence, for our comparision we wrote a simple test application which made a single call to a file `open` API under the target OS and measured the latency in terms of clock cycles before the call and after the return. The instrument did nothing except to measure the latency of instrument invocation and invoking the original construct at the instrumented location (for frameworks that allow such a feature such as Detours and SPiKE). This allowed us to determine both the instrument invocation time as well as the time for the original construct invocation at the instrumented location. As seen from Figure 6h, SPiKE's performance is comparable to other probe-based frameworks, but is not the most efficient. Given that the other probe-based frameworks do not compare in capacity when it comes to malware analysis, the fact that SPiKE can achieve a latency close to these frameworks is acceptable.

6 Background and Related Work

There are various reseach related to the area of instrumentation and dynamic compilation. Broadly, instrumentation can be categorized as source level or binary level. Source level instrumentation basically includes insertion of wrappers in the program source code, which transfer control to the instrument during program execution. Binary level instrumentation on the other hand, accomplish instrumentation without the program sources. To limit our scope of discussion, we concentrate on binary level instrumentation in this section. Binary instrumentation can be categorized into static and dynamic approaches.

Static binary instrumentation is an offline technique that involves rewriting the program binary to insert instrumentation constructs. This art was pioneered by Atom (Srivastava & Eustace 1994), followed by others such as EEL (Larus & Schnarr 1995), Etch (Romer et al. 1997), Morph (Zhang et al. 1997) etc. Static approaches have a serious drawback in that, the tool may not have enough information to deal with mixed code and data within the executable. In the context of malware, static approaches do not suffice as most if not all malware are sensitive to code modification, being self-modifying and or self-checking. Other difficulties with static systems are indirect branches, dynamic libraries and dynamically generated code.

Dynamic binary instrumentation on the other hand involves inserting instrumentation during run-time, addressing the limitations of static approaches. There are two approaches to dynamic instrumentation: probe-based and JIT. Probe-based instrumentation works by dynamically replacing instructions in the target code, with instructions that branch to the instrumented code (jump or trap). Example probe-based frameworks include Dyninst (Buck & Hollingsworth 2000), Dtrace (Cantrill et al. 2004), Detours (Hunt & Brubacher 1999), DProbes (Moore 2001), Linux Trace Toolkit (Yaghmour & Dagenais 2000), Vulcan (Srivastava et al. 2001) etc. There are various drawbacks to probe-based approaches. In the context of malware, probe-based approaches have a severe limitation in that, they cannot cannot be used to probe malware specific functions since most if not all malware are very sensitve to code modification. Also such systems do not observe instrumentation transparency. With recent trend in malware showing increasing anti-analysis schemes, they can be easily detected and countered. Other problems with probe-based approaches are related to arbitrary construct instrumentation on architectures where instruction sizes vary (i.e x86). In such cases, an instruction cannot be replaced with one which is greater than its size, since it would overwrite the following instruction. JIT approaches on the other hand, overcome the transparency problem of probe-based approaches by executing code inside a VM. Examples include Pin (Luk et al. 2005), Valgrind (Nethercote & Seward 2003), DynamoRIO (Bruening 2004), Strata (Scott et al. 2003) and Diota (Maebe et al. 2002). However, in the context of malware, they do not support multithreading and do not carry support for SM-SC code. Also current JIT frameworks are unable to analyze code running in kernel-mode are very slow when compared to their probe-based counterparts. In comparison to the existing frameworks in the area of dynamic binary-instrumentation, SPiKE is unique in that, it is the first instrumentation framework specifically geared to aid in the construction of malware analysis tools. SPiKE provides dynamic coarse-grained binary-instrumentation that it is completely invisible to the target code and cannot be easily detected or countered. The framework has support for multithreading and SM-SC code and can capture code in user- and kernel-mode with minimal latency.

7 Conclusions

We have presented SPiKE, an unobtrusive, efficient, portable, easy-to-use and re-usable dynamic coarse grained binary-instrumentation framework for engineering malware analysis tools. The instrumentation deployed by the framework is completely invisible to the target code and cannot be easily detected or countered. We show that the framework can capture multithreaded and SM-SC code in both user- and kernel-mode while incurring a minimal performance latency. SPiKE currently runs under the Windows and Linux OSs on IA-32 (and compatible) processors. The framework achieves instrumentation using the virtual memory system as a base, that is a commonplace in most platforms. This, coupled with the fact that the SPiKE architecture abstracts platform specific details, enables the framework to be ported to other platforms. We show SPiKEs easy to use APIs enable construction of powerful malware analysis tools with ease and discuss one of our own tools that we have used for behavior monitoring of various malware.

Although there remain other important features of SPiKE for which space does not permit a detailed description (selective instrumentation, support APIs, slicing internals, framework polymorphism etc.), we have shown how SPiKE addresses the shortcomings in current research involving binary-instrumentation in the context of malware. In our belief, the framework is the first of its kind in tailoring an instrumentation strategy conducive to malware analysis. Future works include: (a) raising the stealth levels by developing a supervised code execution environment for privileged malware code, (b) employ a sophisticated code rewriter which would do away with the need for redirection-pads and (c) integrate SPiKE into a full fledged malware analysis environment currently being developed by us.

References

Buck, B. R. & Hollingsworth, J. (2000), An api for runtime code patching, *in* 'Journal of High Performance Computing Applications', Vol. 14(4), pp. 317-329.

Bruening, D. L. (2004), Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Ph.D., M.I.T. (http://www.cag.lcs.mit.edu/dynamorio/).

Cantrill, B. M., Shapiro, M. W. & Leventhal, A. H. (2004), Dynamic instrumentation of production systems, *in* '6th Symposium on Operating Systems Design and Implementation', pp. 15–28.

Hunt, G. & Brubacher, D. (1999), Detours: Binary Interception of Win32 Functions, *in* '3rd USENIX Windows NT Symposium', pp. 135–144.

Intel Corporation. (2003), *IA-32 Intel Architecture Software Developers Manual.*, Vols 1-3.

Larus, J. & Schnarr, E. (1995), EEL: Machine-independent executable editing, *in* 'ACM SIGPLAN Conference on Programming Language Design and Implementation', pp. 291-300.

Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. & Hazelwoo, K. (2005), Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *in* 'ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)', Chicago, IL, USA, pp. 190–200.

Maebe, J., Ronsse, M. & De Bosschere, K. (2002), Diota: Dynamic instrumentation, optimization and transformation of applications, *in* 'Compendium of Workshops and Tutorials held in conjunction with PACT02'.

Microsoft Corporation. (2004), Microsoft Portable Executable and Common Object File Format Specification., Rev.

Moore, R. J. (2001), A universal dynamic trace for Linux and other operating systems, *in* 'FREENIX Track'.

Nethercote, N. & Seward, J. (2003), Valgrind: A program supervision framework, *in* '3rd Workshop on Runtime Verification'.

Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B. & Chen. B. (1997), Instrumentation and optimization of win32/intel executables using Etch, *in* 'USENIX Windows NT Workshop', pp. 1-7.

Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. & Soffa, M. L. (2003), Reconfigurable and retargetable software dynamic translation, *in* '1st Conference on Code Generation and Optimization', pp. 36-47.

Srivastava, A., Edwards, A. & Vo, H. (2001), Vulcan: Binary transformation in a distributed environment, Technical Report MSR-TR-2001-50, Microsoft Research.

Srivastava, A. & Eustace, A. (1994), Atom: A system for building customized program analysis tools, *in* 'ACM SIGPLAN Conference on Programming Language Design and Implementation', pp. 196-205.

Unix System Laboratories. (1998), Tool Interface Standards. ELF: Executable and Linkable Format.

Vasudevan, A. & Yerraballi, R. (2005), Stealth Breakpoints, *To appear in* '21st Annual Computer Security and Applications Conference', Tucson, AZ, USA.

Vasudevan, A. & Yerraballi, R. (2004), SAKTHI: A Retargetable Dynamic Framework for Binary Instrumentation, *in* 'Hawaii International Conference in Computer Science', Honolulu, HI, USA.

Yaghmour, K. & Dagenais, M. R. (2000), Measuring and characterizing system behavior using kernel-level event logging, *in* 'USENIX Annual Technical Conference', pp. 13–26.

Zhang, X., Wang, Z., Gloy, N., Chen, J. B. & Smith, M. D. (1997), System support for automatic profiling and optimization, *in* '16th Symposium on Operating System Principles', pp. 15–26.