Stealth Breakpoints

Amit Vasudevan and Ramesh Yerraballi
Department of Computer Science and Engineering
University of Texas at Arlington
{vasudeva,ramesh}@cse.uta.edu

Abstract

Microscopic analysis of malicious code (malware) requires the aid of a variety of powerful tools. Chief among them is a debugger that enables runtime binary analysis at an instruction level. One of the important services provided by a debugger is the ability to stop execution of code at an arbitrary point during runtime, using breakpoints. Software breakpoints support an unlimited number of breakpoint locations by changing the code being debugged so that it can be interrupted during runtime. Most, if not all, malware are very sensitive to code modification with self-modifying and/or self-checking (SM-SC) capabilities, rendering the use of software breakpoints limited in their scope. Hardware breakpoints supported by the underlying processor, on the other hand, use a subset of the processor register set and exception mechanisms to provide breakpoints that do not entail code modification. This makes hardware breakpoints the most powerful breakpoint mechanism for malware analysis. However, current processors provide a very limited number of hardware breakpoints (typically 2–4 locations). Thus, a serious restriction is imposed on the debugger to set a desired number of breakpoints without resorting to the limited alternative of software breakpoints. Also, with the ever evolving nature of malware, there are techniques being employed that prevent the use of hardware breakpoints. This calls for a new breakpoint mechanism that retains the features of hardware breakpoints while providing an unlimited number of breakpoints, which cannot be detected or countered.

In this paper, we present the concept of stealth breakpoints and discuss the design and implementation of VAMPiRE [1]*, a realization of this concept. VAMPiRE cannot be detected or countered and provides unlimited number of breakpoints to be set on code, data, and I/O with the same precision as that of hardware breakpoints. It does so by employing a subtle combination of simple stealth techniques using virtual memory and hardware single-stepping mechanisms that are available on all processors, old and new. This technique makes VAMPiRE portable to any architecture, providing powerful breakpoint ability similar to hardware breakpoints for microscopic malware analysis.*

1 VAMPiRE is a beast (in folklore) that attacks in a stealth fashion.

1. Introduction

Microscopic malware analysis — a fine-grained analysis process that provides insight into malware structure and inner functioning — helps in gleaning important information regarding a malware to facilitate the development of an antidote. Fine-grained analysis requires the aid of various powerful tools, chief among them being a debugger that enables runtime binary analysis at an instruction level. One of the important services provided by a debugger is the ability to stop execution of code being debugged at an arbitrary point during runtime. This is achieved using breakpoints, which can be of two types: Hardware and Software. Hardware breakpoints, as the name suggests, are provided by the underlying processor and support precise breakpoints on code, data and I/O. They are deployed by programming specific processor registers to specify the breakpoint locations and type. Software breakpoints on the other hand are implemented by changing the code being debugged to trigger certain exceptions upon execution (usually a breakpoint exception).

Software breakpoints support unlimited number of breakpoint locations but suffer from the fact that they modify the target code at runtime. This is clearly unsuitable in the context of malware since most if not all malware possess SM-SC capabilities and are very sensitive to changes made to their code. For example, viruses such as `W32.HIV` [18], `W9x.CIH` [17], `W32.MyDoom` [19] etc. use polymorphic/metamorphic code envelopes and employ a host of integrity checks to detect any changes made to their internal code fragments, to prevent their analysis. Hardware breakpoints on the other hand do not involve any form of code modification and, hence, are the most powerful tool in the repertoire of any debugger tailored for malware. Current processors , however, provide a very limited number of hardware breakpoints (typically 2–4 locations). Thus, a serious restriction is imposed on a debugger to set desired number of breakpoints without resorting to the limited alternative of software breakpoints. Also, with the ever evolving nature of malware, there are techniques being employed that prevent the use of hardware breakpoints to analyze them. For example, the `W32.HIV` virus uses the processor debug registers and the breakpoint exception for its internal computations, thereby effectively thwarting hardware breakpoints. This situation calls for a new breakpoint mechanism that retains the features of hardware breakpoints while providing unlimited

number of breakpoints that cannot be detected or countered.

This paper discusses the concept of *stealth breakpoints* and presents VAMPiRE, a realization of this concept that offers the best of both worlds in the sense of unlimited number of precise breakpoints on code, data and I/O which cannot be detected or countered. This is achieved by employing simple stealth techniques that involve virtual memory, single-stepping and task state segments [2] (for processors supporting legacy I/O) — features found in most new and old processor architectures.

While various ideas using virtual memory for breakpoint purposes have been explored in many debuggers [2, 15, 6], most if not all, allow only data read and/or write breakpoints. Also none of them are specifically tailored for malware analysis and their breakpoint implementation can be easily detected and defeated. To the best of our knowledge, VAMPiRE is the first to combine virtual memory, single-stepping, task state segments (TSS) and stealth techniques to provide a stealth and portable breakpoint framework highly conducive for malware analysis. By *stealth* we mean that the breakpoints inserted using VAMPiRE is completely invisible to the code being debugged. VAMPiRE currently runs under the Windows (9x, NT, 2K and XP) and Linux operating systems (OS) on IA-32 (and compatible) processors and is *portable* on any platform (OS and processor architecture) that supports virtual memory and single-stepping. The framework performance is well within the limits to suit interactive debugging and having a simple and easy-to-use API allows it to be incorporated into existing debuggers with ease.

This paper is organized as follows: In Section 2 we consider related work on breakpoints and compare them with VAMPiRE. In Section 3 we discuss the design and implementation of VAMPiRE. In Section 4 we demonstrate the use of VAMPiRE and present some performance numbers for the framework. We conclude the paper in Section 5 summarizing our contributions with suggestions for future work.

2. Background and Related Work

Breakpoints — debugging aids that provide the ability to stop execution of code at an arbitrary point during execution — are primarily categorized into hardware and software. There are various designs and implementations of breakpoints. Several authors have speculated that efficient data breakpoints require special purpose hardware [20, 4, 11]. There are also surveys [21] which discuss the architectural support towards debugging that emphasize the need for special purpose hardware as a debugging aid. One such mechanism is called ICE breakpoints. ICE or in-circuit emulation is a specialized circuitry embedded within the processors, designed for debugging the internals of the processor. There are various approaches to embed the functionality of in-circuit emulation, in hardware, software and hybrid [5]. ICE breakpoints require supporting hardware [8, 1] and is typically used for processor core debugging than normal program debugging. Many processors also have built-in support for hardware breakpoint facilities. This involves a subset of the processor register set and exception mechanisms to provide precise code, data and/or I/O breakpoints albeit allowing breakpoints to be set on a limited number of locations (typically 2–4 locations).

Software breakpoints on the other hand provide an elegant, cost-effective and scalable solution with the existing hardware. There are various categories of software breakpoints. The first variety relies on program source code availability. This coupled with help from the compiler is used to insert data and/or code breakpoints. Practical data breakpoints [26, 25] use efficient runtime data structures and ideas from compiler optimization to provide several methods of data breakpoints. The method involves checking all read and/or write instructions using complex data flow analysis with a segmented bitmap, reserving registers to hold intermediate values during address lookup.

The second variety of software breakpoints uses processor supported trap and/or breakpoint instructions to set the desired breakpoint. There are a host of implementations as in GDB [15], KDB [3], Windbg [23], DBX [14], WDB [1], Softice [6] etc. In this method, a debugger typically encodes a 1 byte trap instruction at the breakpoint location, while saving the byte that was replaced. When the breakpoint triggers by means of the trap exception, the debugger gets control and the original byte is replaced and the program is restarted to execute the instruction. While this method solves the problem of the number of breakpoints that could be active simultaneously, it does not support data and/or I/O breakpoints and is unsuitable for SM-SC code. Also, there are speculations regarding the correctness of trap-based breakpoint implementations. The combination of trap and single-stepping may result in missed breakpoints in a multithreaded program if not correctly implemented by the debugger [22].

Fast breakpoints [12] suggested a novel way to implement software breakpoints using instruction flow change. The idea is to encode a jump instruction to transfer control to the debugger at the breakpoint. While the idea is similar to that of a trap, this method, to some extent, avoids the problem of correctness of a breakpoint in a multi-threaded program. However, the mechanism supports only code breakpoints and is not applicable to SM-SC code.

There are software breakpoint mechanisms which employ the virtual memory system. Vax debug [2], a source level debugger, implements data read and/or write breakpoints using page protection techniques. The attribute of the memory page containing the breakpoint is modified so as to trigger an exception when any location in the page is accessed. The debugger then performs a few checks to ensure the correctness of breakpoint triggering and responds accordingly. However, this idea was not studied or developed further to study the performance or to support code and/or I/O breakpoints. GDB [15], another popular debugger and Poor mans

2 Task State Segments (TSS) are used to store all the information the processor needs in order to manage a task. This includes the processor registers, stacks, the task's virtual memory mappings etc.

watchpoint [7] also support data breakpoints via a similar mechanism.

Softice [6] supports breakpoint on instruction execution using page protection, but has problems regarding the correctness of the breakpoint. For example, if a code execute breakpoint is set on a branch instruction using page protection, the debugger triggers the breakpoint at the destination address of the branch instruction and not the branch instruction itself. Also, Softice is a commercial product only developed for the Windows OS on IA-32 processors, exploiting the OS internals. Thus, its implementation is not generic and sometimes unstable. As an example, the page protection breakpoint mechanism of Softice is only supported under Windows 9x and not under Windows NT, 2K or XP.

Another category of software breakpoints is found in debuggers based on virtual machines. While complete virtual machine based interfaces such as Simics [16] and debuggers running on them can overcome many problems related to breakpoints and their stealthness, they are seldom constructed in that fashion as they do not specifically target malware analysis and are not well suited for interactive debugging. The alternative of using debuggers based on compatibility layers such as TTVM [13], has its drawback since they do not run on commodity OSs and use offline techniques based on logging, replaying and check-pointing [9]. Also, they only maintain a single or a subset of the execution traces which is unsuitable for SM-SC code.

In comparison, VAMPiRE is a software breakpoint framework — running in real-time on commodity OSs such as Windows and Linux — employing the virtual memory system and page protection techniques similar to the ones used in existing debuggers such as GDB [15], Softice [6] etc., but unique in that: (a) it supports unlimited number of precise code and I/O breakpoints in addition to data breakpoints with low latency, (b) it provides breakpoint persistence similar to hardware breakpoints, (c) it supports SM-SC code and employs stealth techniques that shield the breakpoints from detection and/or countermeasures, and (d) it is constructed to be portable and can be plugged into an existing debugger with ease. These features make VAMPiRE the first, and a very powerful breakpoint framework specifically tailored to combat malware that is increasingly becoming hardened to analysis.

3. Design and Implementation

Our goal in designing and implementing VAMPiRE was twofold. First, it should be able to provide unlimited breakpoints to be set on code, data and I/O with the same precision as hardware breakpoints. Second, the breakpoints should be undetectable and impossible to circumvent. This section describes how VAMPiRE achieves these capabilities. We begin with an overview of the framework. We follow that by a detailed discussion on how VAMPiRE provides stealth breakpoints. Finally, we discuss the framework API.

3.1. Framework Overview

Breakpoints under VAMPiRE are realized through a combination of virtual memory, single-stepping, TSS (for applicable processors) and simple stealth techniques. The basic idea involves breakpoint triggering by manipulation of memory page attributes of the underlying virtual memory system (for code, data or memory-mapped I/O breakpoints) and I/O port access control bits in the TSS (for legacy I/O breakpoints). Note that virtual memory and single-stepping are common to most if not all processor architectures and, a TSS (or an equivalent) is typically found on processors supporting legacy I/O such as the IA-32 (and compatible) processors. Figure 1 illustrates the architecture of VAMPiRE in its current version.

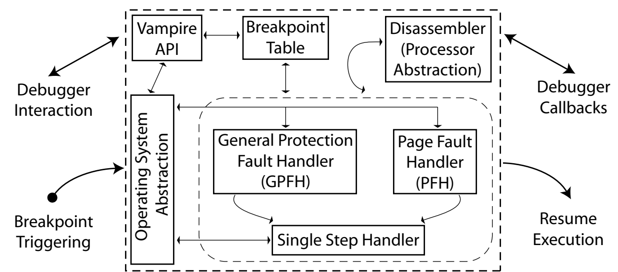

Figure 1. VAMPiRE Architecture

The core of the framework is composed of a *page-fault handler* (PFH), a *general protection fault* handler (GPFH), a *single-step handler* and a *framework API*. The PFH provides breakpoints to be set on code, data and memory-mapped I/O while the GPFH provides legacy I/O breakpoint support. A debugger employing VAMPiRE interacts with the framework through its API to set and/or remove breakpoints at desired locations. Every breakpoint has an associated *callback* (provided by the debugger), a function to which control is transferred upon breakpoint triggering.

When a breakpoint triggers, the framework fault handlers (PFH or GPFH) receive control, determine the type of breakpoint (by employing a *disassembler*) and invoke the callback to do the processing and resume execution. The single-step handler is employed by the framework for breakpoint persistence, a feature that allows a breakpoint to re-trigger automatically in the future. A breakpoint under VAMPiRE can be persistent (re-triggerable) or non-persistent (one-shot) and can be active or inactive at any instant. VAMPiRE uses a *breakpoint-table* — a memory structure specific to the framework — to maintain a list of breakpoints set using the framework. The information stored in the breakpoint-table is used to identify a breakpoint (and its callback) during breakpoint triggering and processing.

3.2. Breakpoint Triggering for Code, Data and Memory-Mapped I/O

VAMPiRE uses the virtual memory system of the underlying platform to provide breakpoints on code, data and

memory-mapped I/O. Support for virtual memory on most processors, is in the form of page directories and page tables. The memory addressing space is divided into chunks of equal size elements called a *page*. (typically 4K or 8K bytes in size). Pages can have various attributes such as `read`, `read/write`, `present`, `not-present`, `user`, `supervisor` etc. These attributes along with a page-fault exception is used to provide virtual memory support and memory protection.

A page-fault exception is generated by the underlying processor when a reference to a page is inconsistent with the page attributes (e.g. a write is issued to a location in the referenced page, while the page attribute is `readonly`). The OS normally installs a handler for the page-fault exception, which implements a paging algorithm, provides protection and other features such as copy-on-write etc. VAMPiRE installs its PFH (pseudo-code shown in Figure 2), replacing and chaining to the existing handler to implement breakpoints on code, data and memory-mapped I/O.

```
 1. obtain linear-address of fault
 2. find memory page corresponding to linear-address
 3. if (no active breakpoints on memory page){
 4.   apply clock patch
 5.   chain to previous page-fault handler
 6. }

 7. mark memory page present preventing recursive faults
 8. find effective address of instruction causing the fault
 9. if (effective address is a breakpoint address){
10.   get breakpoint type (read, write and/or execute)
11.   trigger breakpoint and process
12. }

13. setup re-triggering for persistent breakpoints on
    memory page
14. setup single-step handler for breakpoint persistence
15. apply clock patch
16. return
```

Figure 2. VAMPiRE PFH

To set a breakpoint (code, data or memory-mapped I/O) at the desired memory location, VAMPiRE sets the attribute of the page corresponding to the memory location to `not-present`. This results in a page-fault exception when any location is referenced in that page. When VAMPiRE's PFH gets control it obtains the linear-address of the fault (this is passed to the handler by the processor) and determines the page corresponding to the linear-address. The PFH then performs a check to determine if the page corresponding to the linear-address contains any active breakpoints. If not, the PFH applies a clock patch (if applicable) to hide its processing latency from the malware (see Section 3.6) and chains to the original page-fault handler since the fault is not due to the framework. This is shown in lines 1–6, Figure 2.

If the page corresponding to the linear-address of the fault, contains active breakpoints, the PFH sets the page attribute to `present` to prevent recursive page-faults within the handler. A *disassembler* is then employed to obtain the effective address of the instruction causing the fault. If the effective address is a breakpoint address, the disassembled instruction is analysed to see if it is a read, write or execute breakpoint for code, data and memory-mapped I/O. If so, the breakpoint is triggered and processed. This is shown in lines 7–12, Figure 2.

The PFH then prepares to re-trigger any persistent breakpoints on the page corresponding to the linear-address causing the fault. This is accomplished by populating a global breakpoint re-trigger array (see Section 3.5). A single-step handler is then setup to step over the current instruction that caused the breakpoint to enable breakpoint persistence (see Section 3.4). At this point, the PFH applies a clock patch (if applicable) to hide the latency of the handler from the malware (see Section 3.6). Finally, the PFH returns, marking the end of exception. This is shown in lines 13–16, Figure 2.

3.3. Breakpoint Triggering for Legacy I/O

Legacy I/O uses processor supported I/O instructions to read from or write to a I/O port. Legacy I/O breakpoints involve breakpoint triggering due to such I/O instructions attempting to read from or a write to a specific I/O port. Processors which support legacy I/O along with virtual memory, support legacy I/O protection or virtualization in the form of a TSS (or an equivalent). VAMPiRE relies on the TSS to provide breakpoints on legacy I/O on processors that support them. The TSS consists of a bitmap structure called the I/O Bitmap which is a bit array with 1 bit for every legacy I/O port in the system. If the bit corresponding to a I/O port is set to a 1, the processor causes a general protection fault (GPF) when I/O instructions referencing that I/O port are executed. VAMPiRE makes use of this feature to support legacy I/O breakpoints. It installs its GPFH (pseudo-code shown in Figure 3), replacing and chaining to the existing handler to implement breakpoints on legacy I/O.

```
 1. obtain linear-address of the fault via TSS
 2. disassemble the instruction causing the fault
 3. if (not legacy I/O instruction){
 4.   apply clock patch
 5.   chain to previous GPF handler
 6. }

 7. determine I/O port in disassembled instruction
 8. if (I/O breakpoint on I/O port){
 9.   reset I/O bitmask for the port
10.   find breakpoint type (read or write)
11.   trigger breakpoint and process
12. }

13. setup re-triggering for persistent breakpoints on
    memory page
14. setup single-step handler for breakpoint persistence
15. apply clock patch
16. return
```

Figure 3. VAMPiRE GPFH

VAMPiRE's GPFH semantics is very similar to that of its PFH. To set a breakpoint at the desired I/O location (read or write), VAMPIRE sets the bit corresponding to the I/O port, in the I/O Bitmap array to a 1. This results in a GPF when any access is attempted using that particular I/O port. When VAMPiRE's GPFH gets control, the handler obtains the linear address of the fault via the TSS and disassembles the instruction causing the fault. If the instruction does not belong to the category of legacy I/O instructions, the GPFH applies

a clock patch (if applicable) to hide its processing latency from the malware (see Section 3.6) and chains to the original GPF handler since the fault is not due to the framework. This is shown in lines 1–6, Figure 3.

If the instruction causing the fault is identified as a legacy I/O instruction, the GPFH checks to see if the corresponding I/O port (extracted from the instruction) has a breakpoint set on it. If so, the corresponding I/O permission bit in the I/O Bitmap is set to a 0 to prevent recursive faults from accessing that port within the handler. The GPFH then obtains the breakpoint type (read or write) and processes the breakpoint. This is shown in lines 7–12, Figure 3. The rest of the GPFH processing (lines 13–16, Figure 3) is the same as described for the framework PFH (lines 13–16, Figure 2, Section 3.2) for the framework PFH.

3.4. Breakpoint Persistence via Single-Stepping

The single-step exception is a standard exception on all processor architectures that is triggered upon execution of each instruction when a certain processor flag (also called the trap flag) is activated. VAMPiRE installs its own single-step handler (pseudo-code shown in Figure 4), replacing the existing handler to implement breakpoint persistence. The single-step handler is installed on demand from the framework fault handlers (PFH and GPFH).

When the single-step handler is invoked due to a singlestep exception , the handler first makes sure that any effect of the trap flag (indicator for single-step) is masked out of the instruction that has just been stepped over. This is a very important step towards hiding the framework from the malware being analysed (see Section 3.6). The single-step handler then iterates through every element in the breakpoint re-trigger array (see Section 3.5) and sets the appropriate bit in the I/O Bitmap to 1 (in case of legacy I/O breakpoint) or sets the appropriate memory page attribute to `not-present` (in case of code, data or memory-mapped I/O breakpoint). This ensures that future accesses to the page or the legacy I/O port re-triggers the breakpoint, thereby achieving breakpoint persistence. Finally, the handler resets the trap flag, uninstalls itself, and issues an end of exception. This is shown in lines 1–9, Figure 4.

3.5. Breakpoint Table, Callbacks, Re-trigger Array

VAMPiRE makes use of certain important storage elements for its functioning. Chief among them is the *breakpoint-table*, which is an array of structures, one element for each breakpoint that is set using the framework. The breakpoint-table (shown in Figure 5) consists of (1) the breakpoint address (2) the breakpoint type (code, data, memory mapped I/O or legacy I/O), (3) the breakpoint attributes which include read, write, execute and the persistence flags, (4) the address of the callback which gets control when the breakpoint is triggered, and (5) the breakpoint status, which indicates if the breakpoint is currently active or inactive.

```
1. mask out effect of the trap flag

2. for(each entry in breakpoint re-trigger array){
3.   if (breakpoint type is legacy I/O)
4.     set corresponding bit in I/O bitmap to 1
5.   else
6.     set target memory page attribute to not-present
7. }

8. restore trap flag and uninstall handler
9. return
```

Figure 4. VAMPiRE Single-Step Handler

A *callback* is simply a function that is supplied with parameters identifying the breakpoint address, the breakpoint type (code, data, memory mapped or legacy i/o), the breakpoint condition (read, write or execute) and the breakpoint attribute (persistent or non-persistent). A debugger using VAMPiRE will have to provide such a function that gets control when a breakpoint is triggered to do the appropriate processing. The breakpoint address is the address in memory, which would trigger a breakpoint when accessed. The breakpoint address is the I/O port number if the breakpoint is of type legacy I/O. The breakpoint address and its attributes are usually set by the debugger using the appropriate VAMPiRE API. Though a breakpoint can be set as persistent or non-persistent during its creation, a callback can subsequently over-ride this at runtime by returning an appropriate integer value in the breakpoint attribute parameter, to determine if the breakpoint is to remain persistent or a one shot. One could use different callbacks for different breakpoints, or use a single callback to handle all the breakpoints set using the framework.

Breakpoint Address	Type	Attributes	Callback Address	Status
00501020h	Data	R	812E5000h	✓
10005000h	Code	R W X P	812E5000h	✓
50h	Legacy I/O	R	812E5000h	✗
50001200h	Code	X	812E5000h	✓

R = Read W = Write X = Execute P = Persistent ✓ = Active ✗ = Inactive

Figure 5. VAMPiRE Breakpoint Table

Figure 5 shows several examples of entries one may find in the breakpoint table. The first entry shows a breakpoint set on memory read for the address 00501020h. The breakpoint will only be triggered once since the persistent attribute is missing. Similarly, the second entry shows a breakpoint being set on memory read, write, and/or execute at location 10005000h with the breakpoint being re-triggered automatically because of the presence of the persistent attribute. The third entry shows a legacy I/O breakpoint being set for a read on port 50h that is currently inactive. Also from Figure 5 one can see that all breakpoints are routed through one single callback function at address 812E5000h.

VAMPiRE uses an array, called the *breakpoint re-trigger array* in its fault handlers. Each entry in the breakpoint re-trigger array is a structure containing the breakpoint type, the target page of the breakpoint address (for code, data and

memory-mapped I/O) or the target I/O port number (in case of legacy I/O), and its associated callback address. This array is used by the single-step handler (see Section 3.4) to enable breakpoint persistence.

3.6. Stealth Techniques

VAMPiRE uses the fault handlers (PFH and GPFH) at its core for implementing breakpoints. However, the fault handlers, their latency and issues involving single-stepping can be detected during runtime albeit using some subtle tricks. We now present some simple techniques that the framework employs for stealthness.

The framework PFH (see Section 3.2) the GPFH (see Section 3.3) apply a clock patch during their processing. On the IA-32 (and compatible) processors, the processor maintains a time-stamp counter which is representative of the actual amount of clock cycles that have elapsed since system bootup. The `RDTSC` instruction can be used to obtain the current processor time-stamp counter reading. A malware could use this to detect if its code is being run under any debugger since single-stepping or breakpoints will result in increased latency which is not present during normal execution. Depending on the system VAMPiRE is run under, the framework applies a clock patch resetting the time-stamp counter to a value that mimics the latency close to that of a normal handler. The framework also performs the same reset on the real-time clock since a malware could use the real-time clock to perform a similar latency detection.

VAMPiRE employs a single-step handler for breakpoint persistence. The single-step handler makes use of the processor trap-flag for its functioning (see Section 3.4). The processor usually has instructions which aid in its manipulation. A malware could, for example, on the IA-32 (and compatible) processors use instructions like `PUSHF` and `POP` to check for the trap-flag being active which signifies that its being debugged. In some cases a malware could also employ the single-step exception to perform its functionality. VAMPiRE resolves such issues by masking out the effect of the trap-flag. The framework maintains a per-thread virtual trap-flag status and can simulate a single-step exception by chaining to the original handler in cases where the target malware uses single-stepping. The framework PFH and GPFH install the single-step handler on demand, ensuring that VAMPiRE's single-step handler is always always the first to get control even if a malware installs its own single-step handler in kernel-mode. Note that a malware can install its own PFH and GPFH too, but will have to chain to the default handlers (in this case, VAMPiRE's) since these handlers form the backbone of the host OS.

Another detection scheme against VAMPiRE could employ checking the interrupt vector address range of the PFH and the GPFH (which fall within a constant range for certain OSs regardless of their version), by comparing them with the system default values. Note that a detection scheme cannot check the vector address of the single-step handler since it is installed on demand (see Section 3.4). VAMPiRE uses a retargetable code instrumentation technique [24], to install the exception handlers behind the default handlers. Thus, even though the vector addresses for the various exception handlers will still point towards the system default values, the actual code at these addresses will be modified to execute the frameworks exception handlers. The basic idea behind the instrumentation technique is to alter the target code, in binary, at runtime, in a way so as to execute a replacement code. The replacement code can in turn execute the original target code, if required, in its unaltered form. VAMPiRE employs a random version of the instrumentation technique within the system default handlers such that a malware cannot search for a specific instruction in the code stream of the default handlers to detect the framework (an example would be to detect a branch instruction right at the start of the system handler code or at a specific offset). A debugger using VAMPiRE would ideally load a resident part of itself and initialize the framework right at the boot-stage of the OS, before any device drivers are loaded.

The framework also employs a polymorphic/metamorphic 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 VAMPiRE by searching these elements for a pattern.

A point to be noted is, though VAMPiRE is completely stealth, a debugger incorporating VAMPiRE might not be. For example, if one were to plug VAMPiRE into a debugger such as Windbg [23] or Softice [6], a malware might still be able to detect that it is being analysed. However, this detection is that of the Windbg or Softice and not of VAMPiRE. The authors are currently developing a full fledged stealth debugger codenamed WiLDCAT, which makes use of VAMPiRE to provide features such as selective malware code stream tracing, real-time reverse debugging, memory monitoring and a host of other features in both user- and kernel-mode, making it an indispensable tool for microscopic malware analysis.

3.7. Framework API and Source Organization

The main interface to VAMPiRE is provided in the form of three easy to use API functions: `vampire_init`, `vampire_bpsetup` and `vampire_bpremove`. `vampire_init` is responsible for installing the framework page-fault and the GPF handlers and is invoked only once to initialize the framework. `vampire_bpsetup` is used for setting up a breakpoint at the desired memory or I/O location with a corresponding callback. `vampire_bpremove` is used to remove or deactivate a previously set breakpoint. The API's `vampire_bpsetup` and `vampire_bpremove` rely on the breakpoint table (see Section 3.5), to maintain the current list of breakpoints.

VAMPiRE depends on the host OS to install exception handlers for the page-fault, GPF and the single-step exceptions. This dependency is isolated in the framework source

code, through an OS backend that masks the OS dependent aspects so that it is easy to port VAMPiRE to other OSs with ease. Similarly, the disassembler used by the PFH and the GPFH forms the processor backend that masks the architecture dependent aspects of the framework. Thus, it becomes easy to port VAMPiRE to other architectures as well.

4. Experience and Performance Evaluation

This section will discuss our experience with VAMPiRE and also present the framework's performance measurements. The performance of a breakpoint framework such as VAMPiRE depends on a number of factors, chief among them being the nature of the code being debugged, the dependency of the code on the data accessed, and the style of debugging employed by an individual in terms of setting breakpoints. These factors are not easy to characterize and hence it is difficult to come up with a representative debugging session for performance measurements. This is further complicated by the fact that the same individual can adopt a different style of debugging at any given time for a given code fragment. Therefore, we will concentrate on presenting the performance of VAMPiRE based on debugging sessions with a Windows based virus, `W32.HIV` [18]. The performance of the framework for other debugging sessions can be estimated in a similar fashion. We chose the `W32.HIV` virus for our analysis and discussion since it contains a variety of anti-analysis tricks that one would typically encounter in a malware.

Before we proceed to discuss our experience with the `W32.HIV` virus, a few words regarding the test-bench are in order. To validate VAMPiRE, we have implemented a prototype debugger. The current version of our debugger runs under the Windows OS (9x and XP) on the IA-32 (and compatible) processors. It makes use of VAMPiRE and also has support for traditional hardware and software breakpoints. For test purposes, an AMD Athlon XP 1.8 GHz processor with 512 MB of memory was used. Readings were taken at various points within our debugger after a breakpoint was set and before the triggering of a breakpoint and its processing. We used processor clock cycles as the performance metric for comparison. 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 was used to measure the clock cycles.

The `W32.HIV` is a dangerous per-process memory resident `Win32` subsystem virus that infects Windows executables and MSI (install) archives and corrupts certain important system files. The virus has a unique technique of *upgrading* itself from the internet and also possesses e-mail spreading abilities. Besides employing a stock encryption and polymorphic /metamorphic engine, the virus uses anti-debugging tricks to prevent itself from being analysed. It also has mechanisms of halting the machine completely if a debugger (e.g. Softice, Windbg etc.) is detected in use.

The `W32.HIV` virus and modified strains cannot be analysed using traditional software breakpoints as we describe in the following paragraphs. Even hardware breakpoints fail to help in the complete analysis of the virus. The first part of this section will present the analysis of the virus and the performance measurements of VAMPiRE when compared to hardware breakpoints. This analysis is for code fragments of the virus where both VAMPiRE and hardware breakpoints can be used. The second part of this section will then present an analysis of the virus and the performance measurements of VAMPiRE for code fragments where hardware breakpoints cannot be used thereby showing the utility of the framework.

For purposes of discussion, we will proceed to look at some simplified code fragments of the `W32.HIV` under different debugging sessions with our prototype debugger. The code fragments are shown in the 32-bit assembly language syntax of the IA-32 (and compatible) processors. We have removed details from the code fragments that are not pertinent to our discussion. Consider a code fragment as shown in Figure 6. This code fragment might not look interesting at a first glance. However, after quite a bit of tinkering, it is found that this, and other similar code fragments are in fact examples of the virus polymorphic/metamorphic engine kicking into action. More specifically, during our first encounter with debugging and tracing this code fragment, we had our suspicions on the section of the code fragment with the compare (`CMP`) instructions (lines 13–16, Figure 6), to have something to do with the main branch logic handling the functionality of the virus.

```
                  ...
              1. xor eax, esi
              2. and eax, 38567fffh
              3. add eax, ecx
              4. cmp eax, edi
              5. jbe 10015000
              6. mov edi, eax
              7. mov ecx, [esi+3ch]
              8. xor ecx, [esi+30h]
              9. mov [esi+3ch], ecx
             10. mov esi, [esi+40h]
             11. rep movsb
Breakpoint   12. mov eax, edi
  ---------> 13. cmp eax, 5
             14. je 1001F0F0
             15. cmp eax, 10
             16. je 1001F1F0
             17. cmp eax, 4F
             18. je 1001F4F0
                 ...
```

Figure 6. Self-Modifying Code Fragment

However, when we set a traditional software breakpoint on the second `CMP` instruction (line 13, Figure 6) and let the code fragment execute, the breakpoint is not triggered. Tracing back a few instructions manually, we find the problem to be the section of the code fragment (lines 6–11, Figure 6), that generates new code (at runtime) overwriting existing code starting from the second `CMP` instruction (line 13, Figure 6). This explains why the traditional software breakpoint was not triggered, as it was overwritten by the newly generated code. A manual workaround to this problem is achieved by tracing through the `REP MOVSB` instruction (line 11, Figure 6) one step at a time and inserting a traditional software breakpoint after the new code is generated. However, since a majority of the virus code is littered with such code frag-

ments, this process soon becomes a tedious task.

Now, consider the code fragment as shown in Figure 7, which will be used to demonstrate the self-checking nature of the virus. The W32.HIV has self-checking abilities. This means that the virus has the capability to detect any kind of modification done to its code and will act in a way that will confuse the individual debugging the virus. The code fragment shown in Figure 7 is very well constructed to mislead someone who is trying to study the behaviour of the virus. Though, at a first glance it is very similar to the self-modifying code fragment as shown in Figure 6, it is just a plain sequence of instructions. In any case, when we set a traditional software breakpoint on the second CMP instruction (line 19, Figure 7) — to study the branching in detail — and let the code execute, the breakpoint triggers successfully. Everything appears normal, and there is no sign of any code changes etc. However, when we trace further, we arrive at a section of code that is garbage and leads to spurious faults. The problem is traced back to the section of the code fragment as shown in lines 11–18, Figure 7.

```
                  ...
               1. xor eax, esi
               2. and eax, 38567fffh
               3. add eax, ecx
               4. cmp eax, edi
               5. jbe 100F5000
               6. mov edi, eax
               7. mov ecx, [esi+3ch]
               8. xor ecx, [esi+30h]
               9. mov [esi+3ch], ecx
              10. mov esi, [esi+40h]
              11. xor eax, eax
              12. mov dl, [edi]
              13. or dl, [esi]
              14. movzx edx, dl
              15. add eax, edx
              16. inc edi
              17. inc esi
 Breakpoint   18. loop 12
  ----------> 19. cmp eax, 5
              20. je 100FF0F0
              21. cmp eax, 10
              22. je 100FF1F0
                  ...
```

Figure 7. Self-Checking Code Fragment

Here we see an integrity check being attempted on the code starting from the CMP instruction on line 19, Figure 7. Setting a traditional software breakpoint on this CMP instruction causes the instruction to be overwritten with a breakpoint instruction. However, this behaviour of traditional software breakpoints causes the code fragment to compute a wrong checksum during the integrity check (lines 11–18, Figure 7). Further, the virus is very intelligent in that, it does not terminate or branch to any location upon a failed integrity check. Instead, the checksum is the branch variable (stored in register EAX) itself. In other words, the virus uses the checksum as a representative of the target address of a branch that performs some processing pertaining to the functionality of the virus. Thus, on an incorrect checksum, it branches to a location where the code is nothing but garbage and on a valid checksum the code fragment performs the function it was designed for.

The manual workaround in this case is achieved by precomputing the checksum and adjusting the value in the EAX register and tracing on. But, as with the self-modifying code fragments, there are many such self-checking code fragments. To further complicate matters, the self-checking code fragments are generated at runtime with predefined values for the branch variable (EAX) for each fragment. This predefined value is the value for which the fragment will perform the desired function. If, for a given self-checking code fragment, the branch variable (EAX) contains a different value, the functionality is undefined. This makes the process of precomputing and manually adjusting the value of the EAX register a tedious process.

As seen, traditional software breakpoints fail to help in analysing the virus because of its self-modifying and self-checking abilities. However, we can use VAMPiRE and hardware breakpoints to set breakpoints on code fragments such as the ones shown in figures 6 and 7. The performance measurements comparing hardware breakpoints and VAMPiRE are shown in Figure 8, for various such code fragments under our prototype debugger. With VAMPiRE, for purposes of measurement, code being debugged and breakpoints fall under two categories. The first is when the code being debugged and the breakpoints are on separate memory pages. The second is when the code and the breakpoints are on the same page. The latency due to our framework for both cases with various self-modifying and self-checking virus code fragments is shown in Figure 8. The x-axis is the amount of clock cycles that have elapsed between executing the code with the breakpoint set and breakpoint triggering. The y-axis (category axis) represents the code fragments which were chosen arbitrarily from our debugging sessions with the W32.HIV. Also, a part of the graph is magnified (indicated by dotted lines) to provide a clear representation of categories with low values on the x-axis.

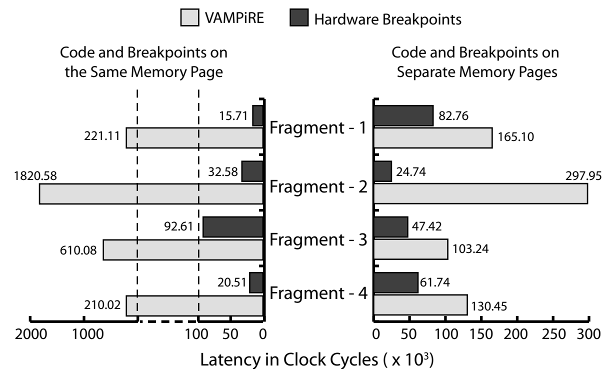

Figure 8. Performance of VAMPiRE vs Hardware Breakpoints

From the graph we can see that the performance of VAMPiRE is comparable to hardware breakpoints in the case where code and breakpoints exist on different memory pages. However, in some cases (as exemplified by Fragment-2, Figure 8, for code and breakpoints on different memory pages) when the data and/or code reference to a page is much higher,

we find that the latency of VAMPiRE is much more than that of hardware breakpoints. When we look at code and breakpoints existing on the same memory page, the latency of VAMPiRE is inherently higher due to its design and the single-step mechanism coming into effect (see Section 3.4). Thus we can conclude that the performance of the framework is highly dependent on the nature of code being debugged and the nature of the breakpoints set by an individual. However, the measurements give a gross indication of the latency one might expect. As seen from Figure 8, the latency of the framework is found to be well within the limits to suit interactive debugging.

While it is possible to set hardware breakpoints on the code fragments discussed so far, we soon find that the virus has efficient anti-debugging capabilities that prevent the use of hardware breakpoints too. Before we discuss these capabilities a few words on hardware breakpoints are in order. The hardware breakpoint mechanism on the IA-32 (and compatible) processors makes use of a set of debug registers named `DR0`–`DR7` and an exception handler for breakpoint and single-step. Registers `DR0`–`DR3` are 32-bit registers which contain the linear address in memory where a code, data or a I/O breakpoint should occur, providing 4 breakpoints in total to be active at a time. `DR7` is a control register which is used to setup the breakpoints initially. When a breakpoint is triggered, the exception handler is invoked with certain status flags in the `DR6` (status) register which describes the type of breakpoint and its attributes (code /data / I/O, read/write/execute).

Let us look at Figure 9 which shows a code fragment from the virus in the context of hardware breakpoints from a debugging session under the Windows 9x OS. The `W32.HIV` spawns (and eventually kills) several ring-0 (privileged) threads in the system at arbitrary intervals. The code used to spawn these threads are concealed within the virus polymorphic/metamorphic layers. The threads are different each time they are spawned, but have a common functionality as shown by the code fragment in Figure 9.

```
                  ...
               1. sub esp, 8
               2. sidt [esp]
               3. mov eax, [esp]
               4. mov edx, [esp+2]
               5. mov eax, [edx+8]
               6. cmp eax, [esi+2bh]
               7. jb 11
               8. cmp eax, [esi+2dh]
               9. ja 11
              10. jmp 12
BADBEHAVE --> 11. jmp CA50D000h
              12. mov dr0, edi
              13. mov eax, esi
              14. ror eax, 16
              15. mov dr1, eax
              16. mov ebx, dr2
                  ...
```

Figure 9. Anti-Debugging Code Fragment

The threads first check to see if the default single-step handler has been altered by checking the system interrupt descriptor table (IDT). It makes use of the `SIDT` instruction to grab hold of the single-step handler address. It then checks if the handler is pointing to something different from the expected system default range (which for Windows 9x is C0000000h–C000FFFFh). If the handler address is out of range, the thread knows that there is a debugger installed, and behaves in a very unfriendly manner (using the `JMP` instruction) to throw the system into an unstable state as long as the debugger is active. This is shown in lines 1–11, Figure 9 (the actual behaviour is concealed behind the `JMP` instruction and is not shown here).

The second level of anti-debugging within the threads employ the debug registers themselves for computation and functioning as shown in lines 12–19, Figure 9. The only workaround to this second level, when using hardware breakpoints, is to manually compute the values in registers and trace them one at a time, while changing the instructions to hold registers other than debug registers. However, this is more than a mission, as (1) there are multiple threads one needs to change that are spawned and killed at regular intervals and (2) the threads themselves have self-checking code fragments, which means one has to set the instructions back to what they were before proceeding to trace further.

However, since VAMPiRE does not make use of any hardware specific debugging registers, it is possible to set breakpoints and trace through such code fragments. Figure 10 shows the performance measurements that were obtained using VAMPiRE alone for such code fragments, using our prototype debugger. The x-axis is the amount of clock cycles that have elapsed between executing the code with the breakpoint set and breakpoint triggering. The y-axis (category axis) represents the code fragments which were once again chosen arbitrarily from our debugging sessions with the `W32.HIV`. A point to be noted is that the graph is not a comparison (since hardware breakpoints cannot be used to analyse such code fragments) but is only provided for the sake of completeness of performance evaluation.

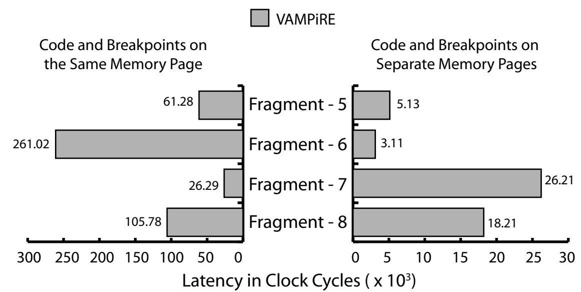

Figure 10. Performance of VAMPiRE

As seen, the features provided by standard debuggers in using hardware and software breakpoints do not suffice to study code employing self-modification, self-checking and/or any form of anti-debugging as in the case of the `W32.HIV` and other similar viruses and malicious programs. With VAMPiRE however, this task is greatly simplified. The framework allows unlimited number of breakpoints to be set

on code, data, and I/O with the same precision as that of hardware breakpoints. The breakpoints set by VAMPiRE cannot be detected or countered and the latency of the framework is well suited for interactive debugging as seen from its performance measurements. These features make VAMPiRE the first and a very powerful breakpoint framework specifically tailored for malware analysis.

5. Conclusions

This paper presented VAMPiRE, a breakpoint framework that overcomes the limitations of hardware and software breakpoints in the context of malware. VAMPiRE cannot be detected or countered and provides unlimited number of breakpoints to be set on code, data, and I/O with the same precision as that of hardware breakpoints. This makes VAMPiRE higly conducive for microscopic analysis of current generation malware, that are increasingly becoming hardened to analysis. We discussed our experience using VAMPiRE to analyze a real world malware, demonstrating the utility of the framework. We also showed that the performance of the framework is well suited for interactive debugging. VAMPiRE can be plugged into any debugger with ease and supports various platforms with its retargetable design. The framework currently runs under the Windows and Linux OSs on the IA-32 (and compatible) processors. VAMPiRE forms the backbone of our malware analysis environment currently under development, and this will provide a completely stealth analysis environment to analyze and combat malware. VAMPiRE is currently known to have only one drawback in that of not supporting legacy I/O breakpoints in kernel-mode. We are working towards a solution to support this feature in the near future.

References

[1] American Arium. Itanium ITP: Intel itanium ICE kit.

[2] B. Beander. Vax debug: An interactive, symbolic, multilingual debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, 1983.

[3] P. A. Buhr, M. Karsten, and J. Shih. Kdb: A multithreadeddebugger for multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and Distributed Tools*, January 1996.

[4] T. Cargill and B. Locanthi. Cheap hardware support for software debugging and profiling. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[5] H. Chen, C. Kao, and I. Huang. Analysis of hardware and software approaches to embedded in-circuit emulation of microprocessors. In *Proceedings of the 7th Asia Pacific Conference on Computer Systems Architecture*. Australian Computer Science Communications, January 2002.

[6] Compuware Corporation. Debugging blue screens. *Technical Paper*, September 1999.

[7] M. Copperman and J. Thomas. Poor man's watchpoints. *ACM SIGPLAN Notices*, January 1995.

[8] Corelis. EMDT/K5 boundary-scan (JTAG) emulator for the AMD K5 processors.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[10] Hewlette Packard. The WDB debugger. *HP Technical Software*.

[11] M. S. Johnson. Some requirements for architectural support of software debugging. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, April 1982.

[12] P. Kessler. Fast breakpoints: design and implementation. In *Proceedings of the conference on Programming Language design and implementation*, June 1990.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Usenix Annual Technical Conference - General Track*, April 2005.

[14] M. A. Linton. The evolution of dbx. In *Proceedings of the Usenix Summer Conference*, June 1990.

[15] M. Loukides and A. Oram. Getting to know gdb. *Linux Journal*, 1996.

[16] P. S. Magnusson, M. Christensson, J. Eskilson, G. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer Society*, 2(35):50–58, February 2002.

[17] McAfee Inc. The W9x.CIH virus. *Virus Information Library*, July 1998.

[18] McAfee Inc. The W32.HIV virus. *Virus Information Library*, October 2000.

[19] McAfee Inc. The W32.MyDoom virus. *Virus Information Library*, February 2004.

[20] J. M. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[21] V. Paxon. A survey of support for implementing debuggers. October 1990.

[22] N. Ramsey. Correctness of trap-based breakpoint implementations. In *Proceedings of 21st Symposium on Principles of Programming Languages*, February 1994.

[23] J. Robbins. Debugging windows based applications using windbg. *Miscrosoft Systems Journal*, 1999.

[24] A. Vasudevan and R. Yerraballi. Sakthi: A retargetable dynamic framework for binary instrumentation. In *Proceedings of the Hawaii International Conference in Computer Sciences*, January 2004.

[25] R. Wahbe. Efficient data breakpoints. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, September 1992.

[26] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the conference on Programming Language Design and Implementation*, June 1993.