Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines

Jedidiah R. Crandall Gary Wassermann Daniela A. S. de Oliveira
Zhendong Su S. Felix Wu Frederic T. Chong
University of California at {Davis, Santa Barbara}
{crandall,wassermg,oliveira,su,wu}@cs.ucdavis.edu, chong@cs.ucsb.edu

Abstract

Worms, viruses, and other malware can be ticking bombs counting down to a specific time, when they might, for example, delete files or download new instructions from a public web server. We propose a novel virtual-machine-based analysis technique to automatically discover the *timetable* of a piece of malware, or when events will be triggered, so that other types of analysis can discern what those events are. This information can be invaluable for responding to rapid malware, and automating its discovery can provide more accurate information with less delay than careful human analysis.

Developing an automated system that produces the timetable of a piece of malware is a challenging research problem. In this paper, we describe our implementation of a key component of such a system: the discovery of timers without making assumptions about the integrity of the infected system's kernel. Our technique runs a virtual machine at slightly different rates of *perceived time* (time as seen by the virtual machine), and identifies time counters by correlating memory write frequency to timer interrupt frequency.

We also analyze real malware to assess the feasibility of using full-system, machine-level symbolic execution on these timers to discover predicates. Because of the intricacies of the Gregorian calendar (leap years, different number of days in each month, etc.) these predicates will not be direct expressions on the timer but instead an annotated trace; so we formalize the calculation of a timetable as a weakest precondition calculation. Our analysis of six real worms sheds light on two challenges for future work: 1) time-dependent malware behavior often does not follow a linear timetable; and 2) that an attacker with knowledge of the analysis technique can evade analysis. Our current results are promising in that with simple symbolic execution we are able to discover predicates on the day of the month for four real worms. Then through more traditional manual analysis we conclude that a more control-flow-sensitive symbolic execution implementation would discover all predicates for the malware we analyzed.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection–*invasive software*

General Terms Security, languages

Keywords worms, malware, virtual machines

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

1. Introduction

The current response when anti-malware defenders discover new malware is to carefully analyze it by disassembling the code, and then release signatures and removal tools for customers to defend themselves from new infections or to remove infections before the malware does any damage. Three trends are challenging this process: 1) increasingly, malware is installing itself into the kernel of the system where analysis is more difficult; 2) malware is becoming more difficult and time-consuming to analyze because of packing (compressing or obfuscating a file so that it must be unpacked before analysis), polymorphism (encrypting the malware body), and metamorphism (techniques such as binary rewriting that change the malware body without changing its functionality); and 3) malware is expected to spread on a more rapid timescale than ever before in the coming years [46, 47]. Suppose a metamorphic, kernel-rootkit-based worm is released that will spread to hundreds of thousands of hosts in just thirty minutes and then launch a denial-of-service attack on a critical information system such as ATMs, the 911 emergency system, or even the Internet itself [41]. Suppose also that the denial of service attack is easily averted if known about ahead of time. How can we discover this ticking timebomb as early as possible?

We propose a novel automated, virtual-machine-based technique to do exactly that. Given a system that is infected with a piece of malware, we describe a technique that extracts how the system is using special timing hardware such as the Programmable Interval Timer (PIT) to keep track of time and then discovers the trigger time for any anomalous events that the system is counting down to. Our goal is to summarize the timetable of a piece of malware quickly and accurately so that responding malware defenders can decide what the best course of action is. For example, the Sober.X worm [57, W32.Sober.X@mm] was programmed to generate random (but predictable to its author) URLs from which to download new instructions starting on `6 January 2006`. Antivirus professionals were able to de-obfuscate the packed code of Sober.X and determine the URLs and the date on which this would occur months ahead of time. The public web servers that the worm instances would be contacting were notified and were able to block those URLs from being registered. In this paper, we aim at enabling this kind of effective response, but on a shorter timescale to be able to handle rapid malware, by automatically discovering the critical date and time.

1.1 Proposed Approach and Contributions of this Paper

The problem turns out to be more difficult than simply speeding up the system clock and seeing what happens. Any study of behavior and time must account for the complex interactions of behavior and time, such as Lamport's study of distributed systems [33] where

events in such systems were shown to be only partially ordered. In our case, malware's behavior can depend on not only the current absolute time (for example, what date and time is shown on the clock) but also relative time (such as how much time has elapsed since initial infection). And, naturally, the passage of relative time changes what the current absolute time is. As a concrete example of this, the Kama Sutra worm [57, W32.Blackmal.E@mm] deletes files on the victim host on the 3rd day of every month, but it only checks the day of the month 30 minutes after either the initial infection or a reboot of the victim host. Thus if a malware analyzer simply infects a machine with Kama Sutra on the 1st of January and speeds up the clock to compress the next year into an hour, this behavior will not be observed because the check for the day of the month will occur only on the 1st of January and never again.

Simply speeding the system up has other disadvantages as well. First, it requires a much more dramatic perturbation of time than our technique does, making it easy for the malware to detect the time perturbation. Furthermore, if the system is somewhat loaded, as it will be for a worm that spawns possibly hundreds of threads to spread itself, the virtual machine will not perform at a high rate of timer interrupts. Some behaviors may be skipped because the worm will never be scheduled to run during that time window. In addition to not revealing some behaviors, it will also not be able to explain why any behaviors that it does elicit occurred.

We propose a technique that uses temporal search to build the malware's timetable. Our approach is to first discover timers by slightly perturbing time and watching for correlations between the rate of perceived time and the rate of updates to each physical memory location. Then through symbolic execution [23] (to discover predicates) and predicate inversion (to make the infrequent case frequent) we build an *abstract program* of the timekeeping architecture of the system. Both of these steps have been implemented for this paper; the first is automated, except for the discovery of additional dependent timers, and the second is done manually. Once this timekeeping architecture has been identified, placing symbolic execution on any timer that the malware might use, by assigning a symbolic expression for each value read from that location, allows us to discover predicates. This step requires distinguishing malware predicates from regular system predicates on time, which is done manually in this paper. Predicate inversion can then elicit the next behavior of the malware without waiting for its predicate on time to become true. From this point it should be possible to build an *abstract program* of the entire trace between the timer and the predicate to discern the malware's timetable. These steps are also done manually in this paper for real malware to demonstrate their efficacy and identify the inherent challenges. By iterating the last two steps for some arbitrary amount of time into the future it is possible to construct a timetable of the malware's behavior. In future work, a richer model than a linear timetable for time-dependent malware behavior is desirable.

Our main contributions toward such an automated system in this paper are 1) detailed results of timer discovery for both Linux and Windows, without making any assumptions about the integrity of the kernel of the infected host; 2) promising initial results on the possibility of using symbolic execution to discover the predicates based on analysis of six real worms; 3) a formalization of temporal search that accounts for the intricacies of the Gregorian calendar; and 4) discussion of the challenges of fully automating the process along with an adversarial analysis.

1.2 Structure of the Paper

The rest of the paper is organized as follows. Section 2 provides some context for our analysis in terms of being both automated and behavior-based. Then Section 3 gives detailed results on timer discovery for both Linux and Windows. This is followed by Section 4 where we analyze six real worms to show the efficacy of symbolic execution to discover malware predicates on the date and time and discuss the inherent challenges. In Section 5 we formally define the problem of how to solve the annotated traces that lead to malware behaviors predicated on time, and illustrate the basic idea with a walk-through of the Code Red worm. Then a discussion of challenges for future work and an adversarial analysis of temporal search in general, against an attacker that seeks to evade our analysis, is in Section 6. Finally, we present related work (Section 7) and conclude (Section 8).

2. Automated, Behavior-Based Analysis

The work presented in this paper differs from traditional malware analysis techniques in two dimensions: behavior-based vs. appearance-based, and in the level of automation. Cohen [6] differentiates behavior-based virus detection from appearance-based detection (such as modern virus scanners) by saying that behavior-based detection is a question "of defining what is and is not a legitimate use of a system service, and finding a means of detecting the difference." Behavior-based analysis has the same goal as detection. For our work we seek to detect illegitimate use of the special hardware that the system provides for keeping track of the date and time. We assume that the system is infected with malware and we wish to know if that malware is using the timekeeping architecture of the system to coordinate malicious behavior; and if so how it is doing this so that we can discern the malware's timetable. Behavior-based detection and analysis, like appearance-based, was shown to be formally undecidable by Cohen [6], but Szor [48] points out that it is not a requirement for a technique to be applicable to every possible piece of malware, it is sufficient for malware defenders to have an arsenal of techniques, one of which will be a good solution in any particular scenario.

In Section 4 we will discuss in detail our experience and lessons learned in performing behavior-based analysis. A fact that can be either a strength or a weakness of behavior-based analysis, depending on how well it is understood, is that the results of analysis are as much a reflection of the virtual environment as they are of the malware itself. A good analogy is Simon's description of an ant walking along the beach [44]. The ant's complex path, walking over twigs, around steep hills, or along ridges, draws more of its complexity from the beach than from the ant. "An ant, viewed as a behaving system, is quite simple. The apparent complexity of its behavior over time is largely a reflection of the complexity of the environment in which it finds itself [44]." Similarly, we discuss in Sections 4 and 6 how the time-dependent behavior of malware is not in fact always a simple, linear timetable and can be miscalculated if the analysis is not done in a sufficiently complex environment.

The complexity of the environment is also a challenge for automation. Even when not considering an attacker who deliberately tries to evade our temporal search analysis, the two separate processes of discovering predicates on the date and time and then relating those predicates to actual dates and times in the real world are interesting program analysis problems. This is because of the intricate integer calculations and loops involved in computations that are based on the Gregorian calendar. There are seven days in the week for cultural reasons, varying numbers of days in each month because the rates of revolution of the moon around the earth and the earth around the sun are not integer multiples [7], and leap years every four years (except for the first years of centuries that are not evenly divisible by 400) because the spin of the earth is not an integer multiple of the length of a year [7]. Thus our current full-system, machine-level symbolic execution engine, DACODA [10], is able to discover predicates on a system timer when the predicate is on a day of the month (or hour, minute, second, etc.), but in fu-

ture work will need to be more control-flow-sensitive to discover predicates on the month or year. Furthermore, once the predicate is discovered, relating it back to an event in the real world (*e.g.* the 15th of the month in the Gregorian calendar) is not a simple matter of solving an expression but requires a weakest precondition calculation (as described in Section 5).

3. Temporal Search

This section describes how to discover timers in a real system using a virtual machine, even if the kernel's integrity has been compromised, and how to automate this process. This step is important because malware is increasingly being implemented as kernel rootkits, and there have even been proposals of implementing malware as a virtual machine in which the victim operating system executes [25].

3.1 How Time is Measured by a System

Without special hardware a system has only an implicit concept of time. Its operations are sequential and the fact that each operation takes some time to complete before the next can begin can be used to infer the passage of time. However, without detailed performance profiling of the entire system, this is not a precise measurement. Because malware shares the processor with the rest of the system it also relies on special hardware to accurately measure the passage of time. In a virtual machine this special hardware is virtualized and completely controlled by the malware analyzers. Measurements of time external to the system can be modeled in many cases, such as the Network Time Protocol (NTP) server connection by the the Sober.X worm. Modeling any arbitrary kind of external time coordination that a piece of malware might do would have to be the subject of future research.

The simplest example of such special hardware, and the most commonly used for PC systems, is the Programmable Interval Timer (PIT). The PIT uses a crystal-based oscillator that runs at one-third the rate of NTSC television color bursts (or 1.193182 MHz) for historical reasons. The PIT device has three timers: one used for RAM refresh, one for PC speaker tone generation, and a third that can be programmed to interrupt the processor at regular intervals. Modern PC-based operating systems use the third timer as their main timekeeping device. Linux kernel 2.4 and Windows XP both program this timer to interrupt the processor at a rate of 100 Hz, meaning that the PIT interrupt is generated 100 times per second. Linux kernel 2.6 programs it for 1000 Hz, and different versions of Windows range from 64 Hz to 1000 Hz. Other special hardware is available in many PC systems, such as the CMOS real time clock, local APIC timers, ACPI timers, the Pentium CPU's Time Stamp Counter, or the High Precision Event Timer. We only consider the PIT for this work, but other special hardware should be a natural extension. A more comprehensive document on timekeeping in systems and virtual machines is available from the VMware company [49].

From the operating system's point of view, time is kept by adding a constant to a variable once per interrupt. Linux kernel 2.4 adds 10,000 to a microseconds counter that is reset every 1,000,000 microseconds when a seconds counter is incremented. The date is kept as a 32-bit counter of seconds starting from `1 January 1970`. Windows adds a value equal to about 10,000,000 (adjusted to the accuracy of the PIT timer for that particular system) to a 64-bit hectonanoseconds counter that counts hectonanoseconds from `1 January 1601`.

The intervals are trivial to infer based on how the PIT is programmed but the epochs (when the absolute time is counted from, such as `1 January 1970` for Linux) are also needed to relate a counter value to an actual date and time in the real world. When a computer is turned off it keeps the date in a known format in the CMOS, and upon boot this value is read by the operating system to initialize the date and time. This epoch, the time of boot, is the only important one since all measurements of absolute time must be derived from it. Through symbolic execution we can determine how any particular absolute time variable is initialized and use that as the epoch.

We define an *absolute time* as a time that relates to an actual time and date in the real world while a *relative time* is relative to some arbitrary start time. Both Linux and Windows keep a relative time that starts at 0 at boot and is incremented on every PIT interrupt. This variable is called "`jiffies`" in the Linux kernel and is used for relative timing needs such as scheduling timeouts. For example, if a process asks to sleep for 10 seconds and the "`jiffies`" variable at start time is 5555, the process will not be scheduled to run again until the "`jiffies`" variable is greater than or equal to 6555, assuming the PIT is programmed for 100Hz (The actual implementation of timers in Linux is not quite this simple).

The PIT model of Bochs (`http://bochs.sourceforge.net`), the virtual machine we use for our experiments, uses the number of instructions executed to roughly guess when PIT interrupts should be scheduled, but this is adjusted to approximate real time. Periodically, a measurement of real time from the host machine is compared to the number of PIT interrupts in the last interval to adjust and more accurately track real time for the next interval. We define *real time* as the passing of time on the physical host machine (which should nearly mirror the physical wall time in the real world) and *perceived time* as the passing of time as seen by the system emulated by Bochs.

3.2 Symbolic Execution

For symbolic execution and predicate discovery, we use the DACODA symbolic execution engine [10]. Basically, DACODA labels values in memory or registers and then tracks those labels symbolically through operations and data movements throughout the entire emulated Pentium system. It also discovers predicates about that data whenever a control flow decision is predicated on a labeled value. We modified DACODA's source code to also discover inequality predicates through the Sign Flag (`SF`) and Overflow Flag (`OF`), in addition to equality predicates through the Zero Flag (`ZF`).

As an example, suppose a byte is labeled and moved into the `AL` register, the integer 4 is added to it, and a control flow transfer is made predicated on the result being greater than 55.

```
mov     al,[AddressWithLabel1999]
     ; AL.expr <- (Label 1999)
add     al,4
     ; AL.expr <- (ADD AL.expr 4)
     ; /* AL.expr == (ADD (LABEL 1999) 4) */
cmp     al,55
     ; FLAGS.left <- AL.expr
     ; /* FLAGS.left == (ADD (Label 1999) 4) */
     ; FLAGS.right <- 55
jg      JumpTargetIfGreaterThan55
     ; P <- new Predicate(GREATERTHAN ZFLAG.left ZFLAG.right)
     ; Q <- new Predicate(LESSTHANOREQUAL ZFLAG.left ZFLAG.right)
     ; /* P == (GREATERTHAN (ADD (Label 1999) 4) 55) */
     ; /* Q == (LESSTHANOREQUAL (ADD (Label 1999) 4) 55) */
     ; if ((ZF == 0) && (OF == SF)) then AddToSetOfKnownPredicates(P);
     ; else AddToSetOfKnownPredicates(Q);
     ; /* Discover predicate if branch taken */
```

This illustrates how DACODA will discover either the predicate (in prefix notation), "`(GREATERTHAN (ADD (Label 1999) 4) 55)`", or its inverse depending on the result of the conditional check.

3.3 The Basic Idea

The basic idea for discovering timers via virtual machines is that the system has certain counters that will speed up or slow down

when the rate of perceived time within the virtual machine is sped up or slowed down. A timer has the following properties:

1. *It should depend on time*: When the rate of perceived time is sped up or slowed down there should be a corresponding speedup or slow-down of the timer.
2. *It should define a series*: A counter has some operation applied to it that defines a series, for example: "1, 2, 3, 4, ...", or "55, 44, 33, ...", or "10000, 20000, 30000, ...". Timers should be based on such a counter. For our purposes we assume a series to be defined such that each subsequent value is simply the previous value plus or minus a constant.
3. *It could depend on another timer*: An example of this is "`xtime.tv_sec`" which counts seconds in the Linux kernel. It is important for calculating the date throughout the system, and it is only incremented every second when a microseconds timer, "`xtime.tv_usec`", reaches the value 1,000,000 and is reset.

3.3.1 Types of Noise

There are several types of noise that must be filtered out to find the timers.

Performance-based phase behavior: Programs can have certain phase behaviors [42] that cause them to update the same memory or increment the same counter at regular time intervals, even though their timing is based on performance and not on time.

Memory updates independent of state: Many memory locations are updated regularly based on time but do not keep state from one timer interrupt to the next. Examples include local variables and return pointers on the stack while timer interrupts are being handled, as well as pixels on the screen.

Memory updates dependent on state that do not define a series: Some memory locations do keep state but do not define a series. An example is a semaphore.

Delayed interrupt handling and NTP: Interrupt handlers in Linux and other operating systems are often divided into a top half and a bottom half. When an interrupt occurs the top half acknowledges the interrupt and schedules work to be done by the bottom half (this is the opposite of the top half and bottom half in FreeBSD but the idea is the same). The bottom half can be executed later or even skipped. For keeping time in Linux "`jiffies`" is incremented in the top half and then when the bottom half executes the "`jiffies`" counter is compared to "`wall_jiffies`", which is the stored "`jiffies`" from the last bottom half execution. If ticks have been skipped the "`xtime.tv_usec`" variable is incremented for every tick that was skipped. This gives "`xtime.tv_usec`" a non-uniform behavior when the system is busy. Furthermore, the Network Time Protocol (NTP), if enabled, occasionally adds or skips ticks to adjust the "`xtime`" structure to inaccuracies in the PIT timer. These kinds of details in the timekeeping architecture of a system can be viewed as noise.

3.3.2 The Basic Steps

Thus, here are the basic steps we use for finding timers:

1. *Do an update count*: The system is allowed to run for a specified amount of time in a number of different stages (4 stages of about 8 seconds per stage was used for all examples that follow, fewer stages or shorter stages may be possible but was unnecessary for these experiments), each stage with a slightly different rate of perceived time (we perturbed time by as much as 35% for these experiments but much smaller perturbations are also possible as will be explained in Section 3.7). We can implement a basic "tainting" mechanism by marking memory with an idempotent expression that "taints" any other values derived from it. When a physical memory location is updated with untainted data it is tainted, and when it is updated with tainted data a physical-address-specific counter is incremented. Thus any memory locations that may be keeping state are tainted. This filters out memory updates independent of state early on for performance reasons. The physical memory locations whose update rates are most correlated with the rate of perceived time are chosen (the top 100 in Linux and Windows, or any appropriate number to account for the amount of noise in the system).
2. *Use symbolic execution to solve the series*: For each candidate timer, the system is allowed to run and any update to that candidate physical memory location is labeled by DACODA. Whenever labeled data is written to that physical memory location the symbolic expression is checked to determine if it defines a series. This can be done for an arbitrary number of times. In practice ten symbolic checks are enough to determine whether the memory location defines a series or not.
3. *Discover additional dependent timers*: For each discovered timer, we mark it with symbolic execution (all reads from the counter are labeled) and if the same predicate is discovered periodically for some minimum number of times (ten is sufficient) we invert it (true becomes false, and false becomes true) and repeat step 1. For example, when symbolic execution is performed on "`xtime.tv_usec`" the predicate is discovered every $\frac{1}{100}th$ of a second that it is less than 1,000,000. Inverting this predicate makes the infrequent case frequent, causing "`xtime.tv_sec`" to be incremented 100 times a second. This will allow us to discover the additional timer "`xtime.tv_sec`" by repeating step 1. Each series can then be solved to convert its values to real time with simple multiplication.

3.4 Linux Example

The following results were taken from a Red Hat Linux system running Linux Kernel version 2.4.21. The Linux kernel keeps a 64-bit internal timer called "`jiffies`" that starts at 0 at boot and is incremented every PIT timer interrupt. To keep track of the date a structure "{`xtime.tv_sec, xtime.tv_usec`}" is updated every time the PIT timer interrupt bottom half is executed as explained above. For Linux Kernel version 2.4 the PIT timer is programmed for 100 Hz, so an interrupt is generated 100 times per second of perceived time.

3.4.1 Update Count

The first step is to run the physical memory location update count for 4 different stages, each with a slightly different rate of perceived time. Perturbing time is accomplished by biasing Bochs' measurements of real time so that real time will appear to Bochs to be passing at a different rate than it actually is, and Bochs will adjust accordingly. Then we must measure the actual rate of perceived time achieved because it will usually not be exactly what was requested.

Doing the update count produces the following top 100 candidate timers (some redundant entries that do not define a series are left out for brevity):

```
Ranking  Error  Phys. Addr.   Update Counts   Symbol
 0 0.000001332  0027ff31  2038 2303 2547 2820 (init_task_union)
   ...
 3 0.000001955  0027e0fc   677  765  846  937 (init_task_union) *
 4 0.000001955  0027e10c   677  765  846  937 (init_task_union) *
 5 0.000001955  0027e18c   677  765  846  937 (init_task_union) *
 6 0.000001955  00269414   677  765  846  937 (i8253_lock)
 7 0.000001955  00269a20   677  765  846  937 (prof_counter)    *
 8 0.000001955  0027e0f8   677  765  846  937 (init_task_union) *
 9 0.000001955  0027ff45  2031 2295 2538 2811 (xprt_clear_backlog)
   ...
12 0.000001955  0027ff48   677  765  846  937 (xprt_clear_backlog)
```

```
   ...
16 0.000001955  0027ff54   677  765  846  937 (xprt_clear_backlog)
   ...
43 0.000001955  0027ffa4   677  765  846  937 (xprt_clear_backlog)
44 0.000001955  002c1800  1354 1530 1692 1874 (irq_desc)
45 0.000001955  002c1810  1354 1530 1692 1874 (irq_desc)
46 0.000001955  002dcd90   677  765  846  937 (kstat)          *
47 0.000001955  002edb20   677  765  846  937 (bh_task_vec)
   ...
51 0.000001955  002edea8   677  765  846  937 (time_phase)
52 0.000001955  002edec4   677  765  846  937 (jiffies)        *
53 0.000001955  002eef88   677  765  846  937 (timer_jiffies)  *
54 0.000002214  002eded4   683  773  855  946 (xtime.tv_usec)  *
55 0.000002351  0027ff6a  2032 2295 2538 2811 (init_task_union)
   ...
63 0.000004048  0026af78   678  766  848  939
64 0.000004072  0027ff72   678  765  846  937 (init_task_union)
   ...
98 0.000011106  0026af74  2728 3078 3408 3780 (xtime_lock)
99 0.000012059  002ed720   677  766  846  939 (irq_stat)
```

The first column is the ranking by error rate, and the second column is the error rate calculated as explained below. This is followed by the physical address and the four actual update counts from each stage. For clarity we have manually appended the symbol from the Linux kernel symbol table for each memory location. An asterisk next to the symbol indicates that in the next step this memory location will be found to define a series. For the experiment above the respective rates of perceived time to real time were 0.71570, 0.93835, 1.14214, and 1.36502 for the four stages.

Error is calculated as the sum of the square of the differences between the update count in all four stages and the perceived rate of time in that stage (with all update counts normalized to the third stage). The value of the error is not as important as the ranking. We want to find the top 100 candidate timers no matter what their error from the true rate of perceived time is, because the actual value can vary from system to system and also depending on what the system is doing.

3.4.2 Solving the Series

In the next step in our example each of the top 100 candidate timers is executed with symbolic execution to determine if it defines a series. For example, the "`jiffies`" counter is defined by the following series (which is the result of a Pentium increment operation):

```
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
PhysicalMemory[0x002edec4] = PhysicalMemory[0x002edec4] + 1
...
```

The "`xtime.tv_usec`" defines this series:

```
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
PhysicalMemory[0x002eded4] = PhysicalMemory[0x002eded4] + 10000
...
```

A memory location such as "`xtime_lock`", which is a semaphore, can be determined to not define a series by observing the following sequence of symbolic operations:

```
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] + 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] + 1
PhysicalMemory[0x0026af74] = PhysicalMemory[0x0026af74] - 1
...
```

Qualitatively speaking, we are mostly interested in the "`xtime`" structure and the "`jiffies`" counter but the other counters discovered ("`prof_counter`", "`kstat`", members of "`init_task_union`", and "`timer_jiffies`") are also important because they do keep track of time and could be used for such by malware. Adding these to the set of timers on which we do symbolic execution should not have a dramatic effect on the accuracy or performance of predicate discovery because these counters do not appear to be heavily used in predicates under normal operation of the system.

3.4.3 Additional Dependent Timers

After running symbolic execution on all of the timers for a while predicates on a certain timer, "`xtime.tv_usec`", will be seen to repeat regularly at a specific program counter location:

```
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
Predicate: (PhysicalMemory[0x002eded4] <= 999999)
...
```

If we invert this predicate (tell the Pentium emulator that it is true when it is false, and that it is false when it is true through the `OF`, `SF`, and `ZF` flags) and repeat steps 1 and 2 we will discover an additional timer, "`xtime.tv_sec`", which defines the following series:

```
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
Predicate: (PhysicalMemory[0x002eded4] > 999999)
PhysicalMemory[0x002eded0] = PhysicalMemory[0x002eded0] + 1
...
```

A simple calculation reveals that this timer is a 1 Hz timer.

3.5 Time Perturbation in Windows

In Windows XP we found three timers of interest: a "`jiffies`"-like counter, which we will call "`TickCount`", at the linear virtual address `0x8053cfc0` (physical address `0x0053cfc0`) in the Hardware Abstraction Layer (HAL) part of the kernel and two hectonanosecond counters mapped in a structure at linear virtual address `0xffdf0000`. This structure is in fact the `_KUSER_SHARED_DATA` structure that is mapped into the virtual address space of every process in the system. The counter at `0xffdf0014` (physical address `0x00041014`), called "`SystemTime`", is the one that is used to calculate the system time and date when a process calls the `GetSystemTime()` library function or any other library function for retrieving the date and time. Thus nearly all Windows malware to date that has a timetable can be analyzed through symbolic execution on this memory address. The other hectonanosecond counter is "`InterruptTime`" at `0xffdf0008` and is irrelevant for our present purposes.

3.6 Comparing How Timers are Used

Figure 1 shows the number of predicates per second discovered for different timers in Windows and Linux over equivalent durations of real time. Note that all five data series were taken at different times and that the rate of perceived time to real time is different for Windows and Linux. What the graph is intended to show is that some timers have a structure that makes them easier to analyze than others. Windows' "`SystemTime`" counter is checked several times a second in the kernel or in library functions having to do with file accesses ("`SystemTime`" is the timestamp that file creation and modification times are given) but the only predicates in user space below the libraries are the predicates every minute from the clock (a big part of each spike is actually the calculation, in the library code mapped for the clock process, of the day, month, year, hour, etc. based on the "`SystemTime`" counter, the predicates are from while loops such as those shown in Figure 3). The pattern of "`xtime.tv_sec`" from Linux is also very simple, suggesting that temporal search on system times and dates need not be very sophisticated. There are many predicates on "`xtime.tv_usec`" but they are virtually all based on two checks for every PIT interrupt:

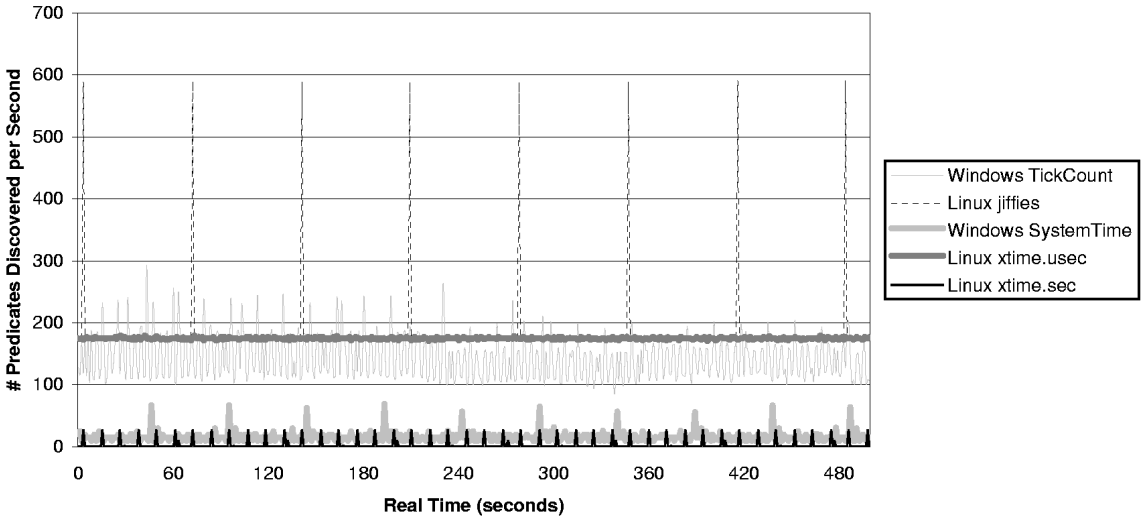

Figure 1. How timers are used.

whether or not it is equal to 999999 (discovered through the ZF flag) and whether or not it is less than 999999 (discovered through the OF flag).

The "jiffies" timer in Linux is slightly more complicated, but with the Linux source code it is easy to determine that all of the predicates on the flat part of the line come from run_timer_list(), which keeps a series of dependent counters that could be discovered in the same fashion as we showed for "xtime.tv_sec", and that the spike every minute comes from only a handful of functions (rt_check_expire_thr(), internal_add_timer(), sys_rt_sigtimedwait(), and rs_timer()). These may or may not be checking predicates for interesting events, such as cron jobs. We would need to understand the "TickCount" timer of Windows as well to be able to, for example, discover a predicate that the Kama Sutra worm is waiting 30 minutes before it checks the date. As shown in Figure 1, however, this may require a great deal of effort or a better understanding of the Windows kernel's timer architecture. Not only does the number of predicates per second vary quite a bit, but analyzing these predicates reveals that they come from a great many different places throughout the kernel and user space.

3.7 Why Must Perceived Time be Perturbed?

In order to filter out performance-based phase behavior noise we need to distinguish between counters dependent on perceived time and counters dependent on performance. Since performance is based on time for a given machine, we need to separate performance and time by perturbing time. Counters dependent on performance should not speed up or slow down when we perturb the rate of perceived time, and we can use this fact to filter them out.

For example, we ran our timer discovery algorithm while the system was busy executing the Kama Sutra worm and correlated with real time, and 202 timers had a smaller error than a block of 14 candidate timers containing the two we were interested in ("SystemTime" at physical address 0x00014014 and "TickCount" at physical address 0x0053cfc0). This was due to performance-based phase behavior noise. Time perturbation of the exact same trace, or correlation with perceived time, moves the block with the two timers we are interested in to the top of the list.

The perturbation of time need not be dramatic. We perturbed time about 10-35% for all experiments in this paper but the perturbation of time need only be slightly larger than the error in the rate of the interesting timers, which is typically never more than 2%. The Bochs PIT model will not allow us to perturb perceived time with that degree of precision.

4. Discovering Predicates

In this section we evaluate the efficacy of discovering malware predicates on a timer using symbolic execution to trace the dataflow from the timer to the predicate. We evaluated six worms using both DACODA [10] and more traditional manual analysis techniques. The relevant timer for all malware presented in this section is "SystemTime" at physical address 0x41014 in Windows XP (0x3cf014 in Windows Whistler, which was used to analyze Code Red).

4.1 Environment

In the explanation of the behavior of each worm it will be apparent why a realistic virtual environment is necessary to produce the desired results. In our environment DACODA runs as a virtual machine implemented as part of the Bochs emulator. For these experiments the emulator ran on a host (192.168.33.1) to which it was connected with the *tuntap* interface (local emulation of an Ethernet connection) and given the IP address 192.168.33.2. Various services were emulated on the host, including TIME (port 37 TCP), NTP (port 123 UDP), HTTP (port 80 TCP), and listening on port 135 TCP to receive Microsoft RPC DCOM connections. The host also ran a Python script using the Scapy library [56] which allowed us to spoof ARP replies, DNS query replies, and TCP reset (RST) packets from unassigned IPs. ARP requests for hosts on the 192.168.33.0/24 network are spoofed with the Ethernet address of the host (192.168.33.1). The host will not reply to DNS queries sent to it intended for fake DNS servers (192.168.33.33 and 192.168.33.44) but the Scapy script also spoofs answers to DNS queries with all queries resolving to 192.168.33.1. For analyzing Code Red and Blaster it was necessary to send TCP RSTs to match outgoing TCP SYN packets so that the worm would continue scanning and not stop to wait for a reply. Sometimes malware expects to be able to contact a static IP address before it will run, which we did not implement because it was not necessary

for any of these six worms. An advantage of Scapy is that any network spoofing necessary can be scripted, usually with only a few lines of Python code.

As a very simple method to distinguish malware predicates from the numerous legitimate predicates in the system, we programmed DACODA to only print predicates below the virtual address `0x40000000` in any process (user code will typically be below this address, libraries and kernel code will be above it). A few predicates on the minute, hour, and day appear every minute from the desktop clock (typically in the lower right-hand corner of Windows systems). Any predicates beyond that came from the malware, which we confirmed by comparing them to published reports and through traditional analysis techniques.

4.2 Code Red v1 (no CME [54] assigned)

We infected a Windows Whistler system (Whistler was an evaluation version of XP) running IIS 5.1 with the Code Red worm (we used the version 1 variant [57, W32/CodeRed.a.worm], which is equivalent to the notorious version 2 variant but did not randomize victim IP addresses). Code Red makes some assumptions about memory locations specific to a particular service pack of Windows 2000 so we helped it find its malicious code on the heap using the virtual machine. We did the experiment between the 20th of February and the 28th of February which is important for understanding the predicates in Figure 4 of Section 5. Since Code Red uses the `GetSystemTime()` library function the dates are in coordinated universal time (UTC) format.

By placing symbolic execution on the "`SystemTime`" timer we discovered two predicates on the date: a comparison to 20 and a comparison to 28. This predicate is checked apparently every time a thread completes a TCP connection to port 80 of a pseudorandom IP address. This was consistent with published reports of Code Red [15, 36], which was programmed to spread until the 20th of the month, perform a denial-of-service attack on the IP address of the White House until the 28th, and then go to sleep for a very long time. Code Red does not predicate any behavior on the month or the year.

It is possible, as we initially did before adding TCP RSTs to the environment, to come to the incorrect conclusion that Code Red only checks the date once (as does Blaster) and therefore once the worm reaches saturation the denial-of-service only occurs upon the re-infection of a machine. With TCP RSTs it is apparent that each Code Red thread checks the date every time it finishes trying to connect to one victim IP address and is about to try another.

4.3 Blaster.E (no CME assigned)

According to published reports on the Blaster worm [57, W32.Blaster.E.Worm] it will perform a denial-of-service on `windowsupdate.com` (the Blaster.E variant we analyzed actually attacks `kimble.org`) if the month is September through December or if the day of the month is the 16th or later. DACODA is only able to discover the predicate on the day of the month, not on the month itself. This is because more control-flow sensitivity would be required to discern the integer relationship between the system timer and the month as calculated in a while loop shown in Figure 4 of Section 5. Figure 4 shows that the integer relationships between the calculated month and year and the timer on which symbolic execution has been placed depend on the conditions of while loops and are not direct expressions, something DACODA does not currently handle.

According to a publicly available decompilation of the Blaster worm [55] it uses `GetDateFormat()` to get the numbers for the day of the month and the month as strings, then converts these to integers. So the "`SystemTime`" timer is converted into the day and month integers (and adjusted to local time), these are converted into strings, and then back to integers before the predicate. This requires DACODA to follow data flow through the Pentium instruction set architecture's address resolution logic, which is also necessary for MyParty.A (but for a different reason).

Published reports [57, W32.Blaster.E.Worm] on the Blaster worm also state that the date is only checked once either upon initial infection or reboot, which we confirmed by discovering the predicate only once even while spoofing TCP RSTs. This kind of behavior is important for malware defenders to understand before responding because it could mean, for example, that slowing down the rate of infection through throttling might exacerbate the denial-of-service attack by causing more initial infections to occur after that critical date.

4.4 Klez.A (no CME assigned)

Klez.A [57, W32.Klez.A@mm] is programmed to infect systems with the ElKern virus, perform large-scale e-mailing, and make files to be zero bytes in length on the 13th of every other month, starting with January. It uses the `GetLocalTime()` library function which adjusts the date and time to the local time zone. The predicate that the month of the year be odd is not discovered by DACODA for the reason already described. The equality predicate for the day of the month to be equal to 13 is discovered. This predicate is repeated periodically meaning that the worm repeatedly checks the date while running.

4.5 MyParty.A (no CME assigned)

MyParty.A only attempts to spread if the month is January, the year is 2002, and the day of the month is between the 25th and the 29th, inclusive. DACODA discovers predicates on the day of the month, but not the month. A predicate against the hard-coded value 2002 is discovered, but is an artifact of various conversions and relating this to a year would require proper tracking of the year through more control-flow-sensitive symbolic execution. We were able to see MyPary.A in unpacked form by placing a breakpoint on the `GetSystemTime()` library function which pauses the worm after it has unpacked itself. MyParty.A uses both the `GetLocalTime()` library function and a combination of `GetSystemTime()` and `GetTimeZoneInformation()` to get the local time in a format broken down into year, month, day of the month, etc. (it is not clear why two equivalent methods are used), and then converts this to an integer (apparently in seconds since 1900). It then takes this integer and breaks it down into year, month, day of the month, etc. before checking the predicate. It also checks the current time against the file creation time of the executable before exiting, which DACODA discovers as equality predicates. Two predicates that are irrelevant but could be useful for identification of MyParty.A is that it checks that the year is between 1970 and 2038. Because of what appears to be compiler optimizations much of this integer arithmetic is done through address resolution logic, which DACODA handles. The predicate is not repeated because MyParty.A always exits in our environment, possibly because no SMTP server was configured.

4.6 Kama Sutra (CME-24) and Sober.X (CME-681)

The Kama Sutra [57, W32.Blackmal.E@mm] worm deletes files on the 3rd day of every month, but only checks the day of the month 30 minutes after either the initial infection or a reboot. The Sober.X worm [57, W32.Sober.X@mm] uses Visual Basic's `DiffDate()` function to calculate the difference in days between the current date and `29 October 2005`. It decides when to start spreading, and on which two days to download new instructions from a public web server, based on this difference being 23, 68, or 69 (the worm also has a condition for -777, which appears to be an error condition). This explains the outbreak on `21 November 2005` and widely publicized updates scheduled for `5 January`

2006 and 6 January 2006 (these may have actually occurred on the 6th and 7th since the logic reportedly is an inequality, see the LURQH analysis [34] for a good explanation).

Sober.X does not use the local system timer but instead contacts a variety of NTP and TIME[1] servers. It keeps a list of the DNS addresses of 40 different servers, so through DNS spoofing we are able to cause the worm to contact the host (192.168.33.1) and then place symbolic execution on the dates and times read over network.

Rather than wait 30 minutes for Kama Sutra to check the date, it should be possible to discover the predicate used to wait 30 minutes and invert that predicate. This would require placing symbolic execution on "TickCount" rather than "SystemTime" and using more sophisticated means of distinguishing the malware predicates from the numerous other predicates in the kernel space on the "TickCount" variable.

Both the Kama Sutra worm and Sober.X are written in Visual Basic. DACODA was not able to discover any useful predicates for either worm. Sober.X uses the Visual Basic DiffDate() function which we suspect would, like predicates on the month or day, require more control-flow sensitivity than is currently implemented in DACODA. Visual Basic represents dates as strings, such as #3/8/2006#. DACODA handles the string conversions of worms written in Visual C++ but apparently needs more work to handle those of worms written in Visual Basic.

4.7 Summary of Results on Discovering Predicates

This raises three challenges that our proposed automated, behavior-based analysis of time-dependent malware must take into consideration. The first is that month and year calculations require control-flow sensitivity. Secondly, the analysis must be able to distinguish between malware predicates and other predicates, possibly by profiling the system before and after infection. Sometimes, we found, the malware also generates predicates that are not relevant to time-dependent behavior. This is due to the fact that some malware uses the system time or other timers as a seed for pseudorandom number generation. And third, the environment must be sufficiently complex for the malware to behave as it would "in the wild."

What becomes apparent in studying the operation of these six worms is that there are many different library calls in Windows that malware can use to check the date and time, and that the format of the date and time can take many various forms, including conversions between UTC and local time. As seen in MyParty.A and Blaster.E, conversions among different formats are often programmed into the malware and not done using existing library functions. Furthermore, in addition to the fact that malware is increasingly executed in kernel space, the _KUSER_SHARED_DATA structure that contains the "SystemTime" timer is mapped into every user space process, so that neither system calls nor library calls are necessary for checking the date and time. This means that, for example, in the context of rapid malware that is obfuscated to make disassembly and manual analysis difficult, even if the attacker makes no specific effort to hide the malware's timetable, by the time a new worm is unpacked and all of its date and time calculations are reverse-engineered, the critical time may have passed.

However, all checks of the date and the time and conversions to different formats can be traced back to the "SystemTime" timer, and ultimately to the PIT; and a suitably control-flow-sensitive, full-system, machine-level symbolic execution implementation should be able to discover and trace any predicates on any form of the date and time from this source, or from other known sources such as NTP.

[1] This is an antiquated protocol on TCP port 37 that returns time as an integer counting seconds since 1900.

$$
\begin{array}{lcl}
\textit{atrace} & ::= & \textit{aentry} \mid \textit{aentry}, \ \textit{atrace} \\
\textit{aentry} & ::= & (\textit{pred}, \ eip, \ n) \mid (\textit{asgn}, \ eip, \ n) \\
\textit{pred} & ::= & \textit{bterm} \mid \textit{bterm} \ \texttt{||} \ \textit{pred} \\
\textit{bterm} & ::= & \textit{bfac} \mid \textit{bfac} \ \texttt{\&\&} \ \textit{bterm} \\
\textit{bfac} & ::= & \textit{bval} \mid \texttt{!}\textit{bval} \\
\textit{bval} & ::= & \textit{comp} \mid (\textit{pred}) \\
\textit{comp} & ::= & < \mid > \mid = \\
\textit{exp} & ::= & \textit{term} \mid \textit{term} \ \textit{op}_a \ \textit{exp} \\
\textit{op}_a & ::= & + \mid - \\
\textit{term} & ::= & \textit{fac} \mid \textit{fac} \ \textit{op}_m \ \textit{term} \\
\textit{op}_m & ::= & \times \mid \div \mid \mod \\
\textit{fac} & ::= & \textit{val} \mid (\textit{exp}) \\
\textit{val} & ::= & \textit{var} \mid i \in \mathbb{Z} \mid \texttt{gettime()} \\
\textit{var} & ::= & u \in V_\tau \mid v \in V_p \\
\textit{asgn} & ::= & \textit{var} \ \texttt{:=} \ \textit{exp}\texttt{;}
\end{array}
$$

Figure 2. Grammar for predicates and expressions, where $eip \in EIP$, $n \in \mathbb{N}$.

5. Recovering the Timetable

Section 4 showed how to discover timers using predicates and dataflow information recovered by a virtual machine. This section presents a technique for using the knowledge of a program's timers along with dynamically discovered predicates to recover the program's timetable. The goal is to relate a predicate, for example, on the day of the month, back to the system timer value after all of the calculations that were performed to calculate the day of the month. Because of the intricacies of these calculations we formalize this as a *weakest precondition* [13] calculation.

5.1 Definitions

Given a piece of time-dependent malware, a malware analyzer would like to know when the program will exhibit malicious behaviors. This section defines a timetable in terms of an execution trace.

By inferring variable names, we construct from DACODA's output for one execution of a program an *annotated trace*, consisting of assignments and predicates. Figure 2 defines annotated traces with a grammar as a list of entries (*aentry*s), each consisting of an assignment or a predicate, followed by the address ($eip \in EIP$ in our grammar, but in practice a tuple of *CR3* and *EIP* is necessary to handle multiple processes in the system) of the instruction in the program code and the time (which we model as a natural number, in the set $\mathbb{N}$, because computers measure time based on discrete events; see Section 3.1) at which the instruction was executed. For two adjacent entries e_1 and e_2 with times t_1 and t_2 respectively, $t_1 \leq t_2$. The predicates in the trace are the branch condition predicates that DACODA discovered as being time-dependent. The function gettime() is an abstract function defined by the results of timer discovery in Section 3. It returns the integer stored in that timer at the time the function is called.

In the formal grammar, *EIP* is a set of addresses, V_τ is a set of variables identified as timers, V_p is a set of variables not identified as timers, and *gettime()* is an abstract function that returns an integer corresponding to the current time.

Executions of a program exhibit *behaviors*. We define behaviors using an analyzer-provided set $L \subseteq EIP$ of *behavior labels*. For example, given a program with a section of code for network activity and a section for waiting, a malware analyzer may label an instruction at the beginning of each section. In our setting, we discover such sections through runtime profiling. Given an annotated trace t, a behavior b of t is a longest subsequence of t such that: the *eip* l of the first instruction is in L, and for an instruction in b with *eip* l', either $l' \notin L$ or $l' = l$. For example, if a

malicious program executes a denial-of-service attack by looping over a section of code that has a single labeled instruction, the whole attack will be considered one behavior. The prefix of t not part of any other behavior is called the *startup behavior*.

A *timetable* summarizes the behaviors of an annotated trace according to time. An entry for a behavior b of a timetable is a triple (τ_s, τ_t, l), where τ_s and τ_t are the time of the first and last instructions in b and l is either the *eip* of the first instruction in b or "startup" if b is the startup behavior. The integer representations of time can be translated into dates, and the *eip*'s can be replaced with labels to produce a more meaningful summary.

5.2 Discovering Timetable Entries

Our goal is to recover a timetable that extends into the future. In order to do this, we need a more efficient method than simply running the program. This section explains how, after the startup behavior has been discovered by observing the program's execution, we discover the end time of a behavior using its beginning time.

If a program will execute a certain behavior for a bounded period of time, it typically runs a loop in which it checks that the time meets some condition, executes some code, and then checks the time again. We can utilize this looping structure to find timetable entries. When label l is first observed, the values of the variables are known. We can then continue to execute the program, recording an annotated trace t of predicates and assignments until the second time l is encountered. At this point, we conclude that we have completed one cycle through the loop, and one of the predicates observed on that cycle is the loop guard. By analyzing t we can recover the number of times this loop will execute, under certain assumptions.

Let $t = [c_0, \cdots, c_i, p, c_{i+2}, \cdots, c_n]$, where p is a predicate and the c's are either assignments or predicates. Because t is one iteration through a loop, we can cut it at p to form $t' = [c_{i+2}, \cdots, c_n, c_0, \cdots, c_i, p]$, an iteration of the same loop in which p is assumed to be the loop guard.

We can now use the *weakest precondition* (WP) [13] of t' to discover the range of possible values for the timer variables at the beginning of t'. A partial correctness assertion on commands is structured as "$\{A\}\ c\ \{B\}$," where A is a predicate on the system state, c is a command that (potentially) modifies the state, and B is the predicate on the system state resulting from the execution of c. A predicate A_1 is weaker than a predicate A_2 if $A_2 \Rightarrow A_1$. The WP, $wp(c, B)$, for command c and post-assertion B is defined as follows (see Dijkstra [13] for more details):

$$wp(v := e, B) = [e/v]B \qquad \text{(assignment)}$$
$$wp(c_1, c_2, B) = wp(c_1, wp(c_2, B)) \qquad \text{(sequence)}$$

where $[e/v]B$ stands for the assertion obtained from B by replacing each occurrence of v with e.

The first-order theory of integers with addition and subtraction is known as Presburger arithmetic, and is decidable [52]. If the only arithmetic operations in the WP are $+$ and $-$, we can use this result to find the ranges of values for the timer variables that satisfy the WP. If arbitrary operations are permitted, this becomes undecidable, but we may apply automated theorem proving techniques from the program verification and automated deduction areas to this setting.

5.3 An Illustrating Example

This section gives an example of how to use the WP semantics to analyze Code Red. Figure 3 shows excerpts from the Sanos `gmtime()` function [37] (Windows source code is not available to us but our symbolic execution results from the Code Red worm in Section 3 confirm that the Windows implementation is similar; note that the epoch for 32-bit UNIX systems is different making the leap year calculation simpler). Figure 4 shows excerpts from a trace that is taken from executing `gmtime()`, and the variable names have been replaced to show the correspondence between the trace and the high-level code. The last line of the trace shows the predicate p that serves as a post-assertion for the trace. For each statement s, if s's WP is different from its post-assertion, then s's WP is displayed above it. The first WP (*i.e.*, the bottom-most shaded predicate) is constructed mechanically using the rules in Section 5.2. The WPs above it are simplified, so that implied predicates are omitted, and arithmetic expressions are simplified. For the mod operation, we identify implied predicates with the rule:

$$\begin{array}{rl} & ((v \texttt{ \% } m \texttt{ = } c_1) \texttt{ \&\& } (v \texttt{ \% } m \texttt{ != } c_2)) \\ \Rightarrow & ((v \texttt{ \% } m \texttt{ = } c_1) \Rightarrow (v \texttt{ \% } m \texttt{ != } c_2)) \end{array}$$

The top-most command uses integer division, in which the remainder gets truncated. To simplify the arithmetic correctly for the expression "`(timer / 86400) < 12478`", we calculate "`timer <` $(($`12478 + 1`$) \times$ `86400) - 1`$)$." Although the operations "`%`" and "`/`" are outside of Presburger arithmetic, we can provide logical inference rules to handle the cases encountered in this example.

The variable `timer` is a known timer variable, so both its frequency and its starting value are known. These values combined with the top-most WP reveal that this path will be taken from 12:00am on `20 February 2006` to 11:59pm on `28 February 2006`.

```
#define LEAPYEAR(year)  ((year) % 4 != 0)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)
const int _ytab[2][12] =
{
  {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
  {31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};
...
while (dayno >= (unsigned long) YEARSIZE(year))
{
  dayno -= YEARSIZE(year);
  year++;
}
while (dayno >= _ytab[LEAPYEAR(year)][tmbuf->tm_mon])
{
  dayno -= _ytab[LEAPYEAR(year)][tmbuf->tm_mon];
  tmbuf->tm_mon++;
}
...
tmbuf->tm_mday = dayno + 1;
```

Figure 3. Excerpt from `ctime()`'s source code.

5.4 Completing the Timetable

Sections 5.2 and 5.3 show how to use the WP semantics to find a range of times in which a code path (on a loop) will continue to be taken. The trace t' (see Section 5.2) is constructed based on an *EIP* $l \in L$, so that every *EIP* of an entry in t' is either l or is not in L. Consequently the time range discovered based on timer variables and a WP defines the time range for a behavior corresponding to l. This behavior can be added to the program's timetable, and if the last two entries have the same label, they can be merged.

In order to begin the next iteration of this process, we set the timer variables to the values we expect them to have at the time immediately after the behavior previously discovered. We then resume execution at the predicate p at the end of t'. By repeating this process, we discover a timetable to an arbitrary point in the future.

```
(timer >= 1077321600) && (timer < 1078185599)
dayno = timer / 86400;
year = 1970;
⋮
(year % 4 != 0)
(dayno >= 365)
dayno >= 12469 && dayno < 12478 && (year % 4 = 2)
dayno = dayno - 365;
dayno >= 12104 && dayno < 12113 && (year % 4 = 2)
year = year + 1;
⋮
(dayno >= 781) && (dayno < 790) && (year % 4 = 0)
(year % 4 = 0)
(dayno >= 366)
(dayno >= 781) && (dayno < 790)
&& (year % 4 != 3) && (year % 4 != 2)
dayno = dayno - 366;
(dayno >= 415) && (dayno < 424)
&& (year % 4 != 3) && (year % 4 != 2)
year = year + 1;
(dayno >= 415) && (dayno < 424)
&& (year % 4 != 0) && (year % 4 != 3)
(year % 4 != 0)
(dayno >= 365)
(dayno >= 415) && (dayno < 424) && (year % 4 != 3)
dayno = dayno - 365;
(dayno >= 50) && (dayno < 59) && (year % 4 != 3)
year = year + 1;
(year % 4 != 0)
(dayno < 365)
tm_year = year - 1900;
tm_yday = dayno;
(dayno >= 50) && (dayno < 59) && (year % 4 != 0)
dayno = dayno - 31;
(dayno >= 19) && (dayno < 28) && (year % 4 != 0)
(year % 4 != 0)
(dayno >= 19) && (dayno < 28)
!(dayno >= 28)
(dayno + 1 >= 20)
tm_mday = dayno + 1
(tm_mday >= 20)
```

Figure 4. Annotated trace with weakest preconditions (shaded). The post-assertion is shown on the last line.

6. Challenges for Future Work

This section enumerates the challenges that must be addressed by future work in this area, both for malware that does not explicitly use knowledge of the analysis to evade analysis, and for malware where the attacker knows about the analysis technique and seeks to evade it. In both bases, we first consider challenges for behavior-based analysis in general and then for temporal search in particular.

6.1 Regular Malware

As discussed in Section 1, it is not necessary for a malware analysis technique to be impossible to evade for it to be useful. In fact, both in practice and in theory, no malware analysis technique is impossible to evade. There still remain challenges for future research even in the domain of regular malware, however, because of the complexity of the domain of the problem we have chosen.

6.1.1 Challenges for Behavior-Based Analysis

Two challenges common to any behavior-based analysis will be 1) defining what is and is not a malicious use of a service; and 2) providing an environment complex enough to elicit the desired behaviors from the malware.

6.1.2 Challenges for Temporal Search

The manifestation of these two challenges for temporal search is particularly interesting. Defining what is and is not a malicious use of the time and date was simple for the six worms analyzed in this paper, but, for malware that installs itself into the system kernel or uses other timers not considered in Section 4, more general techniques are needed. In addition to the need to supply a sufficiently complex environment to elicit time-dependent behaviors from malware, we feel that it will be desirable in future work to develop a formal model of malware behavior over time. The model should be based on formalisms richer than a linear timetable, such as finite state transition systems [5].

A need particular to temporal search is for more control-flow-sensitive symbolic execution, and program analysis techniques specific to the kinds of calculations performed on dates and times. Program analysis involving integer arithmetic is undecidable in general, but date and time calculations are a limited domain in which practical analysis should be possible.

6.2 Evasive Malware

While no malware analysis technique needs to be impossible to evade in order to be useful, it is important that malware defenders know the capabilities and limitations of each technique in their arsenal.

6.2.1 Challenges for Behavior-Based Analysis

The greatest challenge for any behavior-based analysis will be that an attacker with knowledge about the virtual environment that analysis will be performed in can make the malware not behave the same way in that environment as it does on a real victim machine (see [57, W32.Gaobot.EUX] or [57, W32.Toxbot]). For example, the malware might use performance metrics to determine if it is executing in a virtual machine or on native hardware, or it could use the network to find out if it is really connected to the Internet or not. King et al. [25] have explored many of the issues of virtual machine detection in their implementation of malware as a virtual machine. The Pioneer project [40] and recent related work [16] are also relevant to this discussion.

A discussion of the different strengths and weaknesses of attack and defense in this domain is well beyond the scope of this paper, but we will point out that many types of malware analysis, such as temporal search, can be carried out on a trace. Thus all that is needed is zero-performance-overhead logging for deterministic replay. We have built a system similar to ReVirt [14], but where all logging and replay of the virtual machine occurs at the architectural level on port I/O and interrupts. In theory, a hardware implementation of this could achieve zero performance overhead.

6.2.2 Challenges for Temporal Search

Specific to our approach, there are many ways for an attacker to evade temporal search. Our timer discovery step assumes a certain structure of timers: that they define a series and the lower granularity timers have a direct dependency on a predicate that DACODA can discover. Breaking this structure will make this step fail. For example, the attacker could take a microseconds timer and pass it through a channel DACODA does not track before comparing it to 1 million and incrementing a seconds counter. These channels are also a problem for the predicate discovery

step. It may also be more difficult to discriminate between valid uses of the timer and malicious uses by an adversarial attacker. Furthermore, program analysis techniques to track predicates back to a timer are formally undecidable in the general case. To evade the analysis in Section 5 (or make the analysis problem much more difficult in practice) the attacker need only use operations outside of Presburger arithmetic, such as multiplication and division. All of this is based on the fact that if the attacker knows the defenses they can defeat it eventually, and if the defender knows the attack beforehand they can defeat it, but is it possible for an attacker to count down to a specific time in a cryptographically secure manner?

In the general domain of temporal search, an attacker could, in theory, keep a counter called a cryptocounter [53] that is cryptographically secure against analysis to determine what its value is. It is not clear if this directly translates into a way to count down to an event without analysis being able to predict that event and its timing. If a cryptocounter is incremented every second, for example, an analyzer could simply increment it at a much faster rate. A cryptocounter bounded by performance would have to be tuned to the slowest machine that the malware might run on. And any cryptocounter based on an additive homomorphism could have larger values than 1 added to it making parallel search on multiple processors possible. This takes us into the realm of time-lock puzzles and time-released cryptography [38], which is an open issue without an external trusted agent.

7. Related Work

In addition to work cited throughout the paper, there is other related work that may be of interest to the reader in the areas of virtual machines, time perturbation, intrusion detection, and malware analysis.

7.1 Virtual Machines

The topic of virtual machines (VMs) has seen renewed interest recently [2, 19, 26, 39, 45, 51]. Although the major original motivation for VM usage was to provide timesharing capabilities for mainframes, today they are extremely suitable for system or application isolation, platform replication, concurrent execution of different operating systems (OS's), system migration, testing of new application or OS features, or as a secure platform for web applications [8], among other uses [45].

7.2 Time Perturbation

Researchers have used time perturbation to study how I/O and other types of performance scale [21,35], or to understand the behavior of a system [20]. Natural skews in a system's clock have been shown to allow for various kinds of remote fingerprinting [29].

7.3 Intrusion Detection

ReVirt [14] allows for full-system, deterministic replay of a system running on top of a user-mode kernel. This can be used to analyze intrusions with a tool such as BackTracker [24, 27]. IntroVirt [22] is a virtual-machine-based system for detecting known attacks by executing vulnerability-specific predicates as the the system runs or replays, to detect attacks in the period between vulnerability disclosure and patch dissemination.

Livewire [18] is a prototype for an architecture using an intrusion detection system (IDS) running separately from the virtual machine monitor (VMM). The host to be monitored (guest OS and guest applications) runs in the VMM, and the IDS inspects the state of the host being monitored. Terra [17] is an architecture for trusted computing based on VMs. Sidiroglou et al. [43] propose an architecture for detecting unknown malware code inside e-mails by redirecting suspicious e-mails to a virtual machine.

Minos [9] is a security-enhanced microarchitecture that was implemented on the Bochs VM. Minos stops remote control data attacks and a VM implementation of Minos has been used for honeypots [11]. DACODA [10] was built as an extension on top of the Minos VM environment and analyzes attack predicates of worm exploits. VMs have also been used to provide scalability for honeynets by allowing several virtual honeypots executing on a single server [12, 30]. Vrable et al. [50] propose a honeyfarm architecture with the goal of considerably improving honeypot scalability.

7.4 Malware Analysis

In academia, there is relatively little research in the literature on host-based malware detection and analysis compared to the prevalence of this problem. There have been very interesting studies that use binary analysis to detect obfuscated malware [3, 4], deobfuscate packed executables [31], or detect rootkits [32]. These kinds of appearance-based analysis techniques are important, but are only half of the picture. In terms of automated, behavior-based analysis the only two studies that we know of are fairly recent [1, 28]. Both are based on detecting spyware by its spyware-like behavior.

8. Conclusions

We have demonstrated how to use a virtual machine to discover system timers without making assumptions about the integrity of the kernel, and presented promising results on real malware showing that malware timebombs can be detected with symbolic execution. We have also explored the problem domain of temporal search and presented formalisms that account for the intricacies of the Gregorian calendar. We believe that the novel view of this paper, focused on temporal search, of how virtual machine-based analysis can be used to detect malware timebombs shows that behavior-based analysis of malware in virtual machines will be a promising area of research in the coming years.

9. Acknowledgments

We would like to thank our shepherd, Steven Gribble, as well as the anonymous reviewers for their helpful comments. We would also like to thank Tim Sherwood for reading a draft of the paper, Francis Hsu and Yanyan Yang for sharing malware samples with us, Juan Lang for insightful discussions about detecting VMs, and Vern Paxson for pointing out that our initial conclusion about Code Red was incorrect.

References

[1] K. Borders, X. Zhao, and A. Prakash. Siren: Catching evasive malware (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[2] P. M. Chen and B. D. Noble. When Virtual is Better than Real. *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2001.

[3] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. *USENIX Security Symposium*, pages 169–186, August 2003.

[4] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] F. Cohen. Computer viruses: Theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.

[7] N. Copernicus. *On the Revolutions of Heavenly Spheres*. (Available from Prometheus Books, Amherst, New York), 1543.

[8] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy*, 2006.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[10] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. *12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[11] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *DIMVA*, 2005.

[12] D. Dagon, X. Qin, G. Gu, W. Lee, J. B. Grizzard, J. G. Levine, and H. L. Owen. Honeystat: Local worm detection using honeypots. In *RAID*, pages 39–58, 2004.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[15] eEye Digital Security. Advisories and Alerts: .ida Code Red Worm, July 2001.

[16] J. Franklin, M. Luk, J. McCune, A. Seshadri, A. Perrig, and L. van Doorn. Remote virtual machine monitor detection. Presented at the ARO-DARPA-DHS Special Workshop on Botnets, June, 2006.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Network and Distributed System Security Symposium*, 2003.

[19] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. *Tenth Workshop on Hot Topics in Operating Systems (HotOS)*, June 2005.

[20] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd annual International Symposium on Computer Architecture*, 2005.

[21] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time warped network emulation. In *ACM Symposium on Operating Systems Principles*, 2005.

[22] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *ACM Symposium on Operating Systems Principles*, 2005.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[24] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM Symposium on Operating Systems Principles*, 2003.

[25] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing malware with virtual machines. In *IEEE Symposium on Security and Privacy*, 2006.

[26] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *USENIX Security Symposium*, 2003.

[27] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts through Multi-Host Causality. *Network and Distributed System Security Symposium*, February 2005.

[28] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Usenix Security Symposium*, 2006.

[29] T. Kohno, A. Broido, and kc claffy. Remote physical device fingerprinting. In *IEEE Symposium on Security and Privacy*, 2005.

[30] C. Kreibich and J. Crowcroft. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004.

[31] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[32] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. *20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100, 2004.

[33] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[34] LURHQ Threat Intelligence Group. Key Dates in Past and Present Sober Variants. `http://www.lurhq.com/soberdates.html`.

[35] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[36] D. Moore, C. Shannon, and J. Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the Internet Measurement Workshop (IMW)*, 2002.

[37] M. Ringgaard. Sanos source, 2002.

[38] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Cambridge, MA, USA, 1996.

[39] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer Society*, 38(5):39–47, May 2005.

[40] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *ACM Symposium on Operating Systems Principles*, 2005.

[41] R. Sherwood, B. Bhattacharjee, and R. Braud. Misbehaving TCP Receivers can Cause Internet-wide Congestion Collapse. *12th ACM Conference on Computer and Communications Security (CCS)*, 2005.

[42] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[43] S. Sidiroglou, J. Ioannidis, A. D. Keromytis, and S. J. Stolfo. An Email Worm Vaccine Architecture. *ISPEC*, 2005.

[44] H. A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge, MA, USA, 1996.

[45] J. E. Smith and R. Nair. *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[46] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *WORM '04*, pages 33–42, New York, NY, USA, 2004. ACM Press.

[47] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *In Proceedings of the USENIX Security Symposium*, pages 149–167, 2002.

[48] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[49] VMware. Timekeeping in VMware Virtual Machines.

[50] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. *ACM Symposium on Operating Systems Principles*, 2005.

[51] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Rethinking the Design of Virtual Machine Monitors. *IEEE Computer*, 38(5):57–62, May 2005.

[52] P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints (extended abstract). In *Static Analysis Symposium*, pages 21–32, 1995.

[53] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley Publishing, Inc., 2004.

[54] Commmon Malware Enumeration (CME) (Home Page). `http://cme.mitre.org/`.

[55] "Decompiled Source For Ms Rpc Dcom Blaster Worm". `http://www.governmentsecurity.org/archive/t4726.html`.

[56] Scapy. `http://www.secdev.org/projects/scapy/`.

[57] Symantec Security Response - search for malware description. `http://securityresponse.symantec.com/`.