Tick Tock: Building Browser Red Pills from Timing Side Channels

Grant Ho
Stanford University

Dan Boneh
Stanford University

Lucas Ballard
Google

Niels Provos
Google

Abstract

Red pills allow programs to detect if their execution environment is a CPU emulator or a virtual machine. They are used by digital rights management systems and by malware authors. In this paper we study the possibility of browser-based red pills, namely red pills implemented as Javascript that runs in the browser and attempts to detect if the browser is running inside a virtual machine. These browser red pills can limit the effectiveness of Web malware scanners: scanners that detect drive-by downloads and other malicious content by crawling the Web using a browser in an emulated environment. We present multiple browser red pills that are robust across browser platforms and emulation technology. We also discuss potential mitigations that Web scanners can use to thwart some of these red pills.

1 Introduction

Red pills are pieces of code designed to detect if they are being run inside a Virtual Machine (VM) or a CPU emulator[1]. Red pills have several applications such as:

- Honeypot evasion [14]: Since honeypots often run in a virtual machine [15, 1, 10], red pills can help malware evade honeypots: whenever malware detects CPU emulation in its execution environment, it can decide not to infect the host. This makes it harder for honeypots to detect the malware.

- Digital Right Management (DRM): DRM-enabled systems, such as a digital book reader or a movie player, can use red pills to ensure that DRM protected content will not play inside a VM.

Previous work on red pills [6, 18, 14, 3, 5] develop red pills for native software, namely programs that run directly on the operating system. Some use low-level operations such as examining the TLB size; some examine the contents of the operating system's registers and program counters; and some use CPU cycle counters to detect a VM-induced slow-down. The general consensus is that Virtual Machine Monitors (VMMs) are designed for efficiency, not transparency [6]. That is, VMMs generally do not attempt to hide their existence from specifically crafted code designed to detect them.

Our contributions. In this paper we develop browser-based red pills: red pills implemented in Javascript that run in the browser. The interest in browser-based red pills is a result of several systems that attempt to detect malicious content rendered in webpages. These systems run a browser in a CPU emulator or VM and crawl the Web looking for malicious sites. The existence of browser-based red pills can hamper these scanning efforts since a malicious website can choose to withhold malicious content if it detects (from the browser) that it is in an emulated environment. This work is intended to alert Web scanning services to potential limitations of their scanning techniques.

The challenge in designing browser-based red pills is that the browser sandbox is a limited computing environment: Javascript cannot access the TLB, cannot read instruction counters, and cannot look at registers. Thus, many of the techniques available to application-level red pills simply do not work in the browser. The main tool left at our disposal is the browser's timing features, which have a granularity of a millisecond.

Results. We show that timing variations in standard browser operations can reveal the existence of an emulation environment. By comparing the time that the browser takes to perform a simple baseline operation to the time the browser takes to do I/O, spawn Web workers, render complex graphics, or write to local storage, we are able to detect emulation environments with high statistical confidence. Not only do our red pills work against any combination of the two latest versions of Windows and three major browsers, but they also remain effective both against VMs that use binary translation for virtualization and against VMs that use hardware-assisted virtualization.

We describe the implementation of multiple browser red pills in Section 2 and present our evaluation results in Sections 3. We discuss potential defenses against these red pills in Section 4 and conclude with a survey of related work.

[1] The name red pill comes from the movie, "The Matrix".

2 Browser Red Pills

Attack model: We consider a malicious website, `evil.com`, that tries to redirect normal users to a malicious webpage, but evade detection by redirecting honeypots to a benign webpage. When `evil.com` is loaded on an unknown machine, it will execute our browser red pills to determine if it is being executed in a VM (honeypot). We say that our browser red pills are effective if `evil.com` can successfully redirect a VM to a benign website and a normal user to a malicious website.

There are numerous practical implementations of this attack model. For example, consider a scheme where `evil.com` executes our browser red pills and then sends the results to its server through an HTTP GET request. The server can then process the results of the GET request and return a response that redirects the visitor to a benign page if it suspects the visitor is a VM.

We make the assumption that `evil.com` knows the browser and operating system of the visiting machine. Existing literature on browser and device fingerprinting [13, 12, 11] presents a plethora of techniques that easily allow a webpage to determine the visitor's browser and operating system, so we view this as a reasonable assumption.

2.1 Designing Browser Red Pills

While several papers have discussed red pills for native programs (executables that run directly on the operating system), many of these techniques are inapplicable to browser red pills. Prior work on native red pills often relies on detecting low-level anomalies in a system's configuration that are caused by the VM; for example, several prior red pills enumerate the machine's hardware devices, check registers and memory addresses, or look for debugging processes running on the machine to detect the presence of an emulation environment [18, 14, 3]. Unfortunately for attackers, the browser sandbox and Javascript language restrict webpages from the low-level access that these native red pills use. Nonetheless, our work shows that despite these limitations, browser red pills are still possible. By leveraging common Javascript operations, whose execution times are consistently and significantly different in a VM, we are able to construct timing-based red pills that work completely inside of the browser, without the assistance of plugins or browser extensions.

Concretely, our browser red pills work by running two operations on the visitor's machine:

- A "baseline operation", which takes roughly the same amount of time to execute on a normal machine as it does on a virtual machine, and
- A "differential operation", which has a significant and predictable difference in execution time between virtual machines and normal machines.

After executing both operations, the red pills then compute an "adjusted" execution time for their differential operation by dividing the differential operation's execution time by the baseline's execution time. If the expected value for this ratio differs detectably between a virtual machine and a normal machine, then `evil.com` can use the ratio to detect the presence of a VM.

We use a baseline operation in our red pills to account for performance differences that result from different hardware/system configurations and varying background loads on a user's machine. If we used just the raw execution times of our differential operations, it could be unclear whether a longer execution time resulted from older hardware, concurrent activity like watching a video in a separate tab, or being executed in a virtual machine.

2.2 Implementing Browser Red Pills

Each red pill is a snippet of Javascript that executes on a visitor's machine. A red pill executes and times the baseline operation and the differential operation. Afterward, the red pill computes the ratio of the differential operation's execution time to the baseline operation's execution time. The high-level structure of our red pills is shown in Listing 1; while we used the Date object's getTime() function for simplicity, any source of periodic timing can be used for timing measurements (e.g. other Javascript timing objects, implicit wall clocks from periodic functions like requestAnimationFrame, etc.).

```
function redpill() {
   var time = new Date();
   var baseStart = time.getTime();
   baselineOperation();
   var baseTime = time.getTime() - baseStart;

   var diffStart = time.getTime();
   differentialOperation();
   var diffTime = time.getTime() - diffStart;

   return diffTime / baseTime;
}
```

Listing 1: Structure of Browser Operation

We tested two baseline operations and six differential operations for our browser red pills. Our six differential operations fall roughly into three broad categories: I/O operations, threading, and graphics.

For an operation that depends on memory usage or I/O, the size of the operation can have a significant impact on its execution time and the red pill's efficacy; if an I/O operation truly has a difference in execution time

between a VM and a normal machine, a differential operation that executes a greater number of these I/O operations will elicit a larger, and more statistically significant time difference. Consequently, for operations where size/quantity might have a significant impact, we constructed multiple red pills that use several different sizes. When relevant to the operation, we will describe the quantities we used in our different red pills.

2.2.1 Baseline Operations

We tested two baseline operations for our browser red pills: making repeated writes to a DOM node and allocating and deallocating large chunks of memory. These two operations are both simple, common JavaScript operations that had two important "stability" properties on our test machines:

1. Reliability: across multiple executions in a given browser on a given operating system, the baseline operation's execution time stayed approximately the same.

2. Conformity: the baseline operation's execution time on the normal machine took roughly the same amount of time as it did on its corresponding virtual machine.

Because of their stability, pervasiveness, and simplicity, we thought these baseline operations could help account for execution time differences in our differential operations that result from different hardware and background load on real users' machines.

We tested all of our red pills using both baseline operations and report results for both versions of our red pills in Section 3.2; we also conducted experiments that tested the effects of a machine's background activity on our red pills and the utility of our baseline operations, which we also discuss in Section 3.2.

Listing 2 shows how our DOM writing baseline works. First, the DOM baseline operation generates a randomized string of twenty characters; it then creates an empty paragraph node and repeatedly appends this string to the paragraph node's content. We tested this baseline operation at 100, 200, 400, 800, and 1600 repeated writes.

```
function textBaseline() {
   var addString = ``Writes lots of text:"
   addString += Math.floor(Math.random() * 10000);

   var pNode = document.createElement(``p");
   document.body.appendChild(pNode);
   for (var i = 0; i < TEXT_REPETITIONS; i++) {
      pNode.innerHTML = pNode.textContent + addString;
   }
}
```

Listing 2: DOM Baseline Operation

To prevent unexpected browser optimization/string caching when our red pill webpage is reloaded for a new experiment, we randomized the string used in our DOM baseline operation. This helps ensure that repeated trials of our red pills independently perform the full computation for their baseline operation; our data collection/experimental procedures are discussed in greater detail in Section 3.1.

The code for our memory allocation/deallocation baseline is shown in Listing 3. This memory baseline generates a random number and populates an array with instances of the Number class (whose value is set to this random number); immediately after filling this array, we run a loop that pops all the Number objects from the array. Like our DOM baseline, randomness is added between each execution/function call to ensure independence between repeated trials that we performed to collect our results. We tested this baseline operation at 1000, 10000, 20000, 40000, and 80000 allocations and deallocations of Number objects.

```
function memoryBaseline() {
   RANDOM = Math.floor(Math.random() * 1000000);

   var array = new Array();
   for(var i = 0; i < MEMORY_REPETITIONS; i++) {
        array.push(new Number(RANDOM));
   }

   for(var i = 0; i < array.length; i++) {
        array.pop();
   }
}
```

Listing 3: Memory Baseline Operation

To determine the optimal size for these baseline operations, we ran each baseline fifty times for each size on every combination of browser and machine setting that we tested our red pills on (details described in Section 3). We then looked for the sizes that had the smallest variance between multiples runs on a given machine and the smallest difference in execution time between the normal machine and its corresponding VM. This corresponded to 40000 Integer objects for our memory baseline and 400 read/writes for our DOM writing baseline; thus, we used these sizes to construct two versions of all of our red pills, one version for each of the two baselines.

2.2.2 I/O Differential Operations

Console Writing. Our console operation writes the string "Error: Writing to Console!" to the browser's console (which is hidden by default); to test the number of console write operations needed for a stable red pill, we wrote five separate console red pills whose differential operation made 1000, 2000, 3000, 4000, and 5000 consecutive writes to the console. We measured this oper-

ation from before the first write to after the last write; the code for our Console Differential Operation is shown below in Listing 4.

```
function consoleOperation() {
  var error_str = ‘‘Error: Writing to Console!";
  for(var i = 0; i < CONSOLE_REPETITIONS; i++) {
    console.log(error_str);
  }
}
```

Listing 4: Console Differential Operation

Local Storage. HTML 5 introduces the local storage feature, which allows websites to store several megabytes of data persistently on disk. Similar to our console red pills, we created six different versions of local storage red pills. These six versions randomly generate and write a string to local storage for 100, 200, 400, 800, 1600, and 3200 repetitions; each string is 500 character longs. After all strings have been written to local storage, the operation then iterates over the local storage and reads each String back into an array. We measure this operation's time from before the first write (but after all the strings are generated) to after the last string is read from local storage.

2.2.3 Threading Differential Operations

In addition to local storage, HTML 5 enables multithreading capabilities through web workers. Through the web worker API, a webpage can spawn new threads to execute code in Javascript files. The main webpage's thread can then communicate with its web workers (and vice versa) through callback events defined in the web worker API.

Spawning Workers. Our thread spawning operation launches a new web worker to execute a Javascript file. At the beginning of this Javascript file, the web worker immediately gets the current time and sends this time stamp to the main thread. The total execution time is measured from immediately before the web worker is created to the time stamp that the web worker reports when it first begins executing code.

Communicating Between Workers. In addition to measuring the time to spawn threads, we constructed a red pill that measures how long it takes to communicate between two threads (we call this candidate red pill, "rtt operation"). For this rtt operation, we spawn a web worker that gets the current time and sends an "alive" message to the main thread. The main thread then echoes this "alive" message back to the web worker. When the worker receives this echo, it computes the difference between the current time and the time it initially sent the "alive" message; this time difference is then sent back to the main thread as the rtt operation's execution time.

2.2.4 Graphics Differential Operations

Finally, we constructed two more red pills by leveraging the WebGL API, which allows webpages to create complex graphics and games through Open GL.

ReadPixels: CPU - GPU Communication. We constructed a red pill that tests the communication latency between the CPU and graphics card of a website's visitor. This red pill renders and randomly rotates ten triangles with a basic mesh pattern (default shaders and texture) multiple times. After each render call, we used WebGL's readPixels() method to load the pixel bitmap from the visitor's GPU into an array (the visitor's main memory). We tested this differential operation with 40, 80, 160, and 320 render (readPixels) calls. We measured this operation's time from the start of the first readPixels call to after the last readPixels call.

Complex Graphics. Our final red pill tests the speed of the visitor's GPU by rendering lots of polygons with complex shaders and textures. In the background of our canvas, we used the Shader example from Three.js (a popular WebGL library), which renders a large plane that uses a complex whirlpool-pattern shader. On top of the plane, we render three Spheres constructed of many polygons (our Sphere objects were 100 width segments by 100 height segments); to each of these objects, we applied a complex lava texture from Three.js. Our torus and sphere objects were then animated by rotating the objects a random number of radians at each render call. We started the timing measurement at the beginning of the canvas initialization (before any WebGL objects are constructed for rendering) and stopped the measurement after the twentieth animation.

3 Evaluation

3.1 Testing Methodology

We tested our red pills on Chrome version 34, Firefox version 29, and Internet Explorer version 11 on Windows 7 (SP 1) and Windows 8.1. Our Windows 7 host machine used an Intel i5 processor and Intel 4600 integrated graphics card; the Windows 8.1 machine used an AMD A10 processor and AMD Radeon 8750M graphics card. Our virtual machines used identical operating systems and browser versions; we ran the VMs on our Win-

dows 8.1 (AMD) machine using VMWare Virtual Workstation.

Each virtual machine instance was run with 3d graphics acceleration enabled, 2 dedicated processors, and 2 GB of RAM; we believe this is a generous resource allocation when compared to real-world honey pot systems, which may need to run multiple VMs on a single machine in order to operate at scale. We tested our VMs with both hardware-assisted virtualization enabled (Intel VT-X/EPT or AMD-V/RVI mode) and hardware-assistance disabled (binary translation mode); our experiments show that most of our red pills are effective regardless of hardware assisted virtualization. For the rest of this paper, we will refer to the VM setting with binary translation as "VM-BT" and the VM setting with hardware-assisted visualization as "VM-HV".

In total, this setup yielded 18 testing "environments" {Chrome, Firefox, IE} x {Host, VM-BT, VM-HV} x {Windows 7, Windows 8}. To test our red pills, we created a web page that executes a red pill and records the differential operation's execution time, as well as the execution time of both of our baseline operations; one loading of a webpage in a given environment constituted one trial. We conducted one-hundred trials for each environment by reloading each red pill webpage one hundred times, with a 500 ms delay between reloads. Each environment was tested independently (i.e. only one browser and OS [VM or normal machine] was running during each experiment). From these results, we computed the average timing ratio for each differential operation against both of our baseline operations.

To evaluate the efficacy of our red pills, we used unpaired, two-sample t-tests (with $\alpha = 0.05$) to compare the average red pill timing ratio for our normal machine against the average red pill timing ratio for the corresponding VM-BT and VM-HV virtual machines; t-tests are statistical tests used for hypothesis testing. In our case, they test if the distribution of a normal machine's timing ratios is significantly different from the distribution of VM timing ratios. For all red pills that yielded a t-test with p-values less than 0.05 (standard value for a significant difference), we calculated whether one standard deviation away from the mean of the normal machine's red pill ratio was more than one standard deviation away from the mean of the VM's red pill ratio (either VM-HT or VM-BT). If this inequality held, we considered the red pill to be effective because it would allow attackers to set an easy red pill threshold that attacks real users and evades VMs with high probability. Since the distribution of timing ratios in our data seems to approximate the normal distribution, timing ratios that are greater than one standard deviation above the normal machine's mean account for roughly 16% or less of the data (under the normal curve, a single tail above/below the mean constitutes approximately 16% of the probability density). In the context of our experiments, this means that our one-standard-deviation cutoff produced browser red pills that incorrectly attacked a VM or behaved benignly on a normal machine less than 16% of the time.

To summarize, we consider a red pill to be effective if:

1. First, the unpaired t-test (at $\alpha = 0.05$) yielded a p-value of less than 0.05 when comparing the mean of the normal machine's red pill ratio vs. one of the means for a VM's red pill ratio (either VM-BT or VM-HV).

2. Additionally, the red pill satisfied one of these two properties:

   $Mean_{Normal} + SD_{Normal} < Mean_{VM} - SD_{VM}$, if the red pill took longer on the VM.
   $Mean_{Normal} - SD_{Normal} > Mean_{VM} + SD_{VM}$, if the red pill took longer on the normal machine.
   Here, $Mean_{Normal}$ and SD_{Normal} are the mean timing ratio and standard deviation for our normal machine; $Mean_{VM}$ and SD_{VM} have the corresponding definitions for our VM.

3.2 Results

Overview. Tables 1 and 2 present a summary of our red pill efficacy for each environment we tested; Table 1 presents a summary of our red pills that use DOM-writing as their baseline operation and Table 2 presents a summary of our red pills that use memory allocation as their baseline. A suffix of BT means the red pill was effective against a VM using binary translation, and a suffix of HV means that the red pill was effective against a VM with hardware-assisted virtualization enabled. The numbers in each cell represent the p-value obtained from our t-tests that compared the normal machine's average timing ratio against the VM's average timing ratio; a p-value close to zero indicates a high statistical confidence that there is a significant difference between the distribution of normal machine timing ratios and the distribution of VM timing ratios.

Variable Sized Red Pills. Recall that for three of our red pills, "Console Writing", "Local Storage", and "Read Pixels: CPU-GPU", we constructed multiple versions of the red pill that varied the operation size (i.e. number of read/writes); Tables 1 and 2 present the results for the red pill sizes that successfully distinguished between the most environments. For local storage and reading pixels from the GPU, the maximum size we tested against provided strictly more successful red pills than the smaller sizes, so we report the results for 3200 read/writes for local storage and 320 ReadPixels calls in our tables.

DOM Baseline	Chrome	Firefox	IE
Windows 8 BT	Console Writing ($<1.0 \cdot 10^{-6}$) Local Storage ($<1.0 \cdot 10^{-6}$) ReadPixels ($<1.0 \cdot 10^{-6}$) Spawning Workers ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Local Storage ($<1.0 \cdot 10^{-6}$) Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 8 HV	Console Writing ($<1.0 \cdot 10^{-6}$) Local Storage ($<1.0 \cdot 10^{-6}$) Spawning Workers($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 7 BT	ReadPixels ($<1.0 \cdot 10^{-6}$)* Complex Graphics ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	ReadPixels ($<1.0 \cdot 10^{-6}$) Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 7 HV	ReadPixels ($<1.0 \cdot 10^{-6}$)* Complex Graphics ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Local Storage ($<1.0 \cdot 10^{-6}$)* ReadPixels ($<1.0 \cdot 10^{-6}$) Complex Graphics ($<1.0 \cdot 10^{-6}$)

Table 1: Successful Red Pills for DOM Baseline. The rows represent the VM settings (BT is a VM with binary translation and HV is a VM with hardware-assisted virtualization enabled) and each column represents a major browser. The number in parentheses is the p-value for the t-tests that compare normal machine's mean ratio vs. the VM's mean ratio; smaller numbers indicate a higher statistical confidence that there is a difference between normal machine timing ratios and VM timing ratios; for non-zero p-values less than $1.0 \cdot 10^{-6}$, we simply list the value as "$<1.0 \cdot 10^{-6}$". Red pills with an asterisk ran faster on the VM than on the normal machine (i.e. the timing ratio for the normal machine was larger than the VM's timing ratio). Details of these red pills were described in Section 2.3.2.

However, for our console writing red pills, we found that the number of writes, past 2000 console writes, did not affect the red pill's environment coverage (i.e. a red pill that makes 2000 writes to console is effective against Chrome on Windows 8 BT/HV, whereas a red pill that makes 4000 writes to console is still only effective against Chrome on Windows 8 BT/HV).

Deterministic Browser Red Pills. During our experiments, we noticed that Firefox refuses to render WebGL contents in a virtual machine - even though it has no problem rendering the exact same WebGL contents on exactly the same operation system on a normal machine. This provides an easy mechanism to distinguish between Firefox in a VM and Firefox on a normal user's machine. We suspect this is a problem with Firefox's whitelist of acceptable graphics cards for WebGL. When viewing the VM's configuration in Firefox through about:support, we noticed that Firefox reported VMWare's vSGA graphics card as the systems graphics card and disabled WebGL because of "unresolved driver issues". At the same time, Chrome and IE also reported VMWare's virtualized graphics card as the system's graphics card, but they still enabled WebGL features and rendered our WebGL content in the VM. Given this browser reported information, we believe that Firefox's implementation does not support WebGL in VMs. More broadly, this is a good illustration of the difficulties in constructing fully transparent VMs/undetectable honey pots; even virtualization bugs that seem security-irrelevant can leak information that can be used for malicious purposes.

Red Pills that Run Faster on VMs. As noted by asterisks in our results tables, several of our red pills were successful because their timing ratios were significantly larger on the normal machine than on the corresponding VM; in other words, for these red pills, the differential operation ran slower on the normal machine. While we don't have a definitive reason for this surprising result, we believe most of these differences can be attributed to the effects of virtualized hardware on I/O operations. Many of the red pills that run faster on a VM are I/O operations, namely our Local Storage red pills and our ReadPixel red pills that test the communication speed between the CPU and GPU. For both of these red pills, our VMs use some form of virtualized hardware (either a virtualized disk or graphics card). Because the VM is interacting with virtualized devices, I/O operations in the VM might run faster because of optimizations/in-memory caching, or emulation that the VM performs (which can mitigate the number of expensive I/O operations to the actual physical devices). Future work should design experiments to more rigorously and precisely identify the reason why certain red pills run faster on VMs than on their normal machine counterparts.

Effect of Background Activity. To explore the effects of background activity on our red pills, we re-ran all of

Memory Baseline	Chrome	Firefox	IE
Windows 8 BT	Console Writing ($<1.0 \cdot 10^{-6}$) Local Storage ($<1.0 \cdot 10^{-6}$)	ReadPixels (0) Complex Graphics (0)	Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 8 HV	Console Writing ($<1.0 \cdot 10^{-6}$) Local Storage ($<1.0 \cdot 10^{-6}$) Spawning Workers ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 7 BT	ReadPixels ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Local Storage ($<1.0 \cdot 10^{-6}$)* ReadPixels ($<1.0 \cdot 10^{-6}$)* Complex Graphics ($<1.0 \cdot 10^{-6}$)
Windows 7 HV	ReadPixels ($<1.0 \cdot 10^{-6}$)*	ReadPixels (0) Complex Graphics (0)	Local Storage ($<1.0 \cdot 10^{-6}$)* Complex Graphics ($<1.0 \cdot 10^{-6}$)

Table 2: Successful Red Pills for Memory Baseline. The rows represent the VM settings (BT is a VM with binary translation and HV is a VM with hardware-assisted virtualization enabled) and each column represents a major browser. The number in parentheses is the p-value for the t-tests that compare the normal machine's mean timing ratio vs. the VM's mean timing ratio; for non-zero p-values less than $1.0 \cdot 10^{-6}$, we simply list the value as "$<1.0 \cdot 10^{-6}$". Red pills with an asterisk ran faster on the VM than on the normal machine (i.e. the timing ratio for the normal machine was larger than the VM's timing ratio).

the red pills on our normal machines. In this second round of testing, we followed the exact same procedure outlined in Section 3.1, except that prior to visiting our red pill website, we opened three additional tabs in the browser; the first tab played a long Youtube video, the second tab contained the researcher's personal email account, and the final tab contained a popular news website. By running these three tabs in the background, the resulting timing measurements from our normal machines should provide a reasonable approximation of their performance on a real user's machine; we did not load and test our VMs with similar background activity because it is unlikely that a honeypot runs large amounts of background activity during its analysis.

With the exception of our "Spawning Workers" red pill, all of our red pills for both baselines remained effective (based on the same criteria we presented in Section 3.1). Since our "Spawning Workers" red pills only worked for Chrome running on Windows 8 originally, we still have enough red pills to fully cover every combination of operating system, major browser, and virtualization technology; this suggests that our idea of using a baseline operation to scale the red pill timing measurements enhances the robustness and practicality of our browser red pills.

Summary. Overall, our browser red pills fully cover the three most popular browsers on the two latest versions of Windows, even when common browsing activity is concurrently run in the background. Additionally, both baseline operations generated enough red pills to fully cover all these execution environments, which suggests that either baseline can be used for browser red pill constructions. Furthermore, our experiments indicate that even when advanced techniques like hardware-assisted virtualization are used, our browser red pills remain effective at distinguishing a VM from a normal machine. Ultimately, the high statistical confidence and broad effectiveness of our red pills at identifying virtual machines show that browser red pills are possible, despite their inability to access low-level information that native red pills frequently rely on.

4 Red Pill Defenses

In this section, we discuss three potential defenses to browser red pills. While two of these defenses face significant challenges, we believe our third defense will be effective against browser red pills for the time being.

4.1 VM-Obfuscation Defenses

Our first two defenses aim to make a honeypot indistinguishable from a normal user machine. The first defense relies on developing better virtualization technology, while the second defense attempts to distort the timing measurements of red pills.

Fully Transparent Virtual Machines. While numerous advances have been made in virtualization technology, our work presents several browser red pills that work even against VMs with hardware-assisted virtualization; thus, detection frameworks that leverage improved virtualization to disguise their presence still face

challenges in defeating our browser red pills [4]. In order to fundamentally defend against all timing side channels, we would ideally have fully transparent VMs; however, building fully transparent VMs remains an open problem and some researchers believe that building them is fundamentally infeasible [6].

Moreover, even if fully transparent VMs existed, honeypots would still need to hide the overhead incurred by the operational structure of anti-malware organizations. In order to operate at scale, large honeypot systems are unlikely to give a single VM the entire hardware-resource allocation of the underlying normal machine or purchase/use expensive hardware like graphics cards for analysis; these operational overheads are likely to enable reliable timing-based red pills, especially for red pills that rely on heavy computation from expensive hardware, like our graphics red pills. Given the lack of technology that enables fully-transparent VMs and the practically-induced, performance overhead in honeypot systems, we believe that fully-transparent VMs are not a viable defense for large-scale honeypot systems.

Corrupting Red Pill Timing Measurements. If a honeypot can effectively distort the timing measurements used by red pills, it might be able to trick a nefarious server into revealing its malicious content. Three time-distortion techniques come to mind: honeypots could "cheat" on expensive operations to speed up their execution time, add random noise to javascript timing measurements, or add delays to certain operations in order to distort the red pill's timing ratios.

A cheating honeypot might try to speed up expensive operations like rendering graphics by forgoing execution and sleeping for a short amount of time instead. Unfortunately, attackers often have simple ways to check that an operation was actually executed (thereby detecting a cheating honeypot). For example, consider a honeypot that cheats on its graphics operations; rather than rendering graphics, the honeypot simply sleeps for a short amount of time for each graphics operation. In response, an attacker can write WebGL code that renders images that contain several patches of homogeneous color and performs several animations on the images, such as rotations. In addition to retrieving the red pill timing measurements, the malicious website will also fetch pixels at specific locations where the colored patches should be in the final, rotated image. If the rgb values of these chosen pixels don't match the expected result of the animation, the attacker can detect that the unknown visitor is a "cheating" honeypot, who doesn't actually render WebGL graphics.

With cheating ruled out, honeypots might try adding random noise to JavaScript timing measurements to increase the probability that they pass a red pill check. Unfortunately, there are numerous ways for a website to measure time (Date.getTime, Performance.now, setInterval, etc.), which means that a honeypot would need to identify all possible timing sources in Javascript and randomly alter the time returned by each call. Moreover, beyond the various timing API's in Javascript, attackers might be able to construct a variety of implicit timing sources through periodic functions like requestAnimationFrame() or by sending periodic pings to their servers as the red pills execute on the visitor's machine. With this insight, a red pill can defeat a random-noise honeypot by combining multiple sources of timing to detect anomalies in the visitor's reported times. For example, a red pill might measure a differential operation's execution time with both Date.getTime and Performance.now measurements. If the two time source's measurements differ by more than a few milliseconds, then the website can guess that its visitor is a honeypot who adds random noise to time sources. Furthermore, a malicious website could execute its red pills multiple times and average the timing results to cancel the effect of random noise; an attacker might even be able to analyze the timing variance of multiple red pill executions to detect a honeypot that adds random timing noise.

Finally, honeypots can try adding delays to the red pill baseline operations in order to decrease the timing ratios and make it seem like the VM is executing at a normal machine's speed. If we assume that a honeypot can identify a red pill's baseline operations (we discuss this more in the following section), then it can use this information to distort the timing ratios and extract a website's malicious content for more detailed analysis. However, if a honeypot is unable to identify all possible baseline operations, then it needs to commit to frequently adding delays to common JavaScript operations, like writing to the DOM, allocating memory, and any other possible baseline operation. Given that large-scale honeypots need to scan tens of millions of websites, adding even a couple of milliseconds of delay to these common operations could translate to non-trivial losses in scanning throughput. Thus, while adding delays to distort timing ratios might defeat red pills, honeypot operators may need to evaluate whether the lost scanning time justifies the use of this defense.

4.2 Detection Techniques

Given these challenges in making virtual machines indistinguishable from normal machines, we believe that honeypots should focus on identifying the presence of browser red pills or the malicious content that is hidden by the red pills.

Symbolic Execution Techniques. Several papers have been written on extracting malicious behavior from evasive or obfuscated programs using symbolic execution [9], [2]. Unfortunately, these techniques can easily be evaded by webpages that use browser red pills. Unlike native programs, webpages have much greater flexibility with the code they execute and where the code comes from; it is perfectly normal for webpages to send data to other websites and load content/code from many different URLs. To illustrate the challenges of symbolic execution against browser red pills, consider the following scenario: when an unknown user visits `evil.com`, `evil.com` executes our browser red pills, encodes their values as HTTP GET parameters in a URL to its server, and loads the URL in an iframe. Based on the red pill values encoded in the url, the malicious server then chooses to return a benign webpage to be loaded in the iframe if it suspects the visitor is a honeypot. Since the honeypot is never served any malicious code/content, there is nothing for a symbolic execution system to extract.

Detecting the Presence of Red Pills. Rather than trying to extract hidden contents from a webpage, it might be easier to detect the presence of red pills themselves. Two opportunities exist for detecting browser red pills: detecting baseline operations and detecting differential operations.

Currently, our scheme uses two baseline operations, either of which could be used alone to construct browser red pills. Against our memory allocation baseline, we envision using a heap analysis tool to detect unusually large memory operations; already, tools like Nozzle [17] analyze a webpage's memory allocation to detect heap-spraying attacks, so this might be an effective technique against memory-based red pills. Against our DOM writing baseline, honeypots can monitor the DOM calls made by a webpage; since our DOM writing baseline makes hundreds of reads and writes to the DOM in a short time interval, analyzing the frequency of DOM calls might be sufficient to detect our DOM-based red pills. Implicitly, these detection techniques rely on an underlying assumption that a wide variety of stealthier baseline operations do not exist; future work should examine whether a variety of smaller baseline operations can be built from common JavaScript operations. If this cannot be done, then detecting baseline operations can an effective counter measure to browser red pills.

Additionally, honeypots can try to detect the differential operations. Unfortunately, aside from our local storage operation, the differential operations used in our successful red pills might be hard to detect. While our console operation makes thousands of writes to console, this can easily be the result of buggy JavaScript code (which exists en-mass on the web); so unlike detecting our DOM writing baseline, it is unclear how to detect our console writing operation. Against our graphics red pills, it seems unlikely that a honeypot can successfully determine if a graphics operation is malicious in light of the numerous fancy WebGL images and games on the web. Thus, detecting a red pill's differential operation seems like a less effective approach than detecting the baseline operation used by red pills.

5 Related Work

Our techniques for browser red pills relate to three areas of security research: red pills for native programs, methods for honeypot evasion/malicious website cloaking, and web fingerprints for identifying browsers and devices.

Red Pills for Native Programs. As discussed earlier, several papers study red pills for native software (programs that run directly on the operating system) [18], [14], [3], [5]; however, many of the techniques used in these papers do not work for browser red pills. Many of these native red pills are constructed using low-level operations, such as examining the contents of the operating system's registers and program counters [18], [14], and [3]; these tests are unusable for browser red pills because the browser sandbox and language abstractions of Javascript prevent websites from accessing this information. In addition to these low-level, anomaly-detection red pills, Franklin et al. [5] run select operations hundreds-of-thousands of times to create "fuzzy benchmarks" that detect a virtual machine based on performance degradation; however, this approach assumes it has kernel level access and counts performance degradation based on the number of cpu-cycles elapsed, making it unusable for Javascript-based red pills. Finally, for native red pills that don't need low-level or root access, Chen et al. [3] presents a technique using TCP timestamps to detect anomalous clock skews in VMs; but, this technique takes several minutes to execute, making it impractical for malicious web pages (a normal user is unlikely to wait more than a few seconds for a page to load). Moreover, this clock skew technique requires sending streams of hundreds of SYN packets to the VM, which is easy for a honeypot to detect as malicious behavior and hard for an attacker to obfuscate.

Thus, our work is distinct from prior red pill literature because we present the first red pills that run completely within the browser; this more restricted setting has a number of important attack applications, such as web malware that wants to hide zero-day, browser exploits.

Malicious Website Cloaking and Honey Pot Evasion. In addition to the work on detecting evasive malware that we discussed in our Defense section, several papers have studied cloaking/evasion techniques that are currently used in-the-wild by malicious websites.

Rajab et al. [16] discuss different methods that have been used to evade Google Safe Browsing's web malware detection system, as well as a number of defenses and detection enhancements that counter these evasion techniques. Our browser red pills address the more fundamental problem of generally distinguishing a VM from a normal user's machine; additionally, our red pills are harder to defeat than the techniques presented in [16], which use fragile methods like cloaking against Google IP addresses.

Kapravelos et al. [8] also studies evasion techniques that malware-geared honeypots face; however, the evasion techniques they examine rely on vulnerabilities in older browsers (e.g. IE 7), affect only a limited set of custom honeypots, or defer evasion to the malicious, native program that gets executed in the honeypot.

Additionally, several papers study cloaking for blackhat-search engine optimization (SEO) [19], [7]; blackhat SEO is the process of presenting malicious/spam content to normal web users, but a tailored webpage to search engine crawlers that cause the website to earn a high search ranking. In these papers, blackhat-SEO techniques work primarily by simple user-agent cloaking (a malicious webpage checks if the visitor's user-agent claims to be a search engine crawler). These techniques are easily defeated by honeypots that perform user-agent spoofing (and by browser extensions that modify a client's user-agent to look like a search engine crawler); our browser red pills present a more fundamental challenge to honeypots that cannot be easily resolved by spoofing HTTP header information.

Fingerprinting Browsers and Machines. Many papers have been written on how a website can fingerprint not only a visitor's browser, but also the underlying device; these techniques can be used to track a user without the use of any cookies or consent from the user [12], [13], [11]. In general, these fingerprints are constructed by probing the browser's DOM and analyzing the behavior of the browser's Javascript engine to extract details about the browser and the underlying system.

While fingerprinting can be used to distinguish between a real user's browser and a browser emulator, it is unclear how the fingerprints can be directly applied to distinguish a honeypot from a real user's machine. Since honeypots often run a real browser to visit suspicious webpages, there is nothing fundamentally different between a honeypot's browser configurations and a real user's browser configurations. Even if there is a discernible difference between the configurations of a honeypot browser and all normal users' browsers, an attacker would need to know a-priori what the honeypot fingerprint is in order to evade the honeypot; this a-priori knowledge would also be needed for every honeypot system among all anti-malware organizations and may need to be updated for every update/change to a honeypot's browser or system.

Given these challenges, we believe that browser fingerprinting does not offer an easy and fundamental way to evade honeypot analysis; however, browser fingerprints can be combined with our work to effectively evade honeypots. In particular, prior work on fingerprinting offers a litany of techniques that accurately identify both the browser and operating system of a website's visitor. These are two pieces of information that our browser red pills need to effectively detect if a webpage is being loaded in a virtual machine. Thus, the work on browser and device fingerprinting and our work on browser red pills are complementary and address different threat models.

6 Conclusion

Our work shows that despite limitations of the Javascript execution environment, browser red pills are possible. By leveraging the execution times of common Javascript operations, we construct a variety of red pills that work purely within the browser. This shows that malicious web sites can potentially hide browser exploits from honeypot detection. Our empirical evaluation shows that these red pills are effective regardless of the choice of browser on either of the two latest versions of Windows. Furthermore, even when a VM uses hardware-assisted virtualization, our red pills can successfully distinguish the VM from a normal machine. We outlined a few defenses that need to be further investigated in future work. Future work can also explore why certain red pills run faster in VMs and whether browser red pills are actively being used in-the-wild for honeypot evasion.

Acknowledgments

The work is supported by NSF and DARPA. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF and DARPA.

References

[1] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *EICAR*, page 180192, 2006.

[2] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[3] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[4] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[5] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. Van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review*, 42(3):83–92, 2008.

[6] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS*, 2007.

[7] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi. deseo: Combating search-result poisoning. In *USENIX Security Symposium*, 2011.

[8] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from monkey island: Evading high-interaction honeyclients. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 124–143. Springer, 2011.

[9] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 443–457. IEEE, 2012.

[10] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID*, pages 78–97, 2008.

[11] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[12] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. C. Wien. Fast and reliable browser identification with javascript engine fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP)*, volume 5, 2013.

[13] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 541–555. IEEE, 2013.

[14] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proc. of WOOT'09*, pages 2–2, 2009.

[15] N. Provos. A virtual honeypot framework. In *USENIX Security Symposium*, pages 1–14, 2004.

[16] M. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in circumventing web-malware detection. *Google, Google Technical Report*, 2011.

[17] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[18] J. Rutkowska. Red pill ... or how to detect VMM using (almost) one CPU instruction. `www.hackerzvoice.net/ouah/Red_%20Pill.html`.

[19] D. Y. Wang, S. Savage, and G. M. Voelker. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 477–490. ACM, 2011.