Hiding Debuggers from Malware with Apate*

Hao Shi
USC/Information Sciences Institute
4676 Admiralty Way Ste 1001
Marina del Rey, CA 90292, USA
haoshi@usc.edu

Jelena Mirkovic
USC/Information Sciences Institute
4676 Admiralty Way Ste 1001
Marina del Rey, CA 90292, USA
mirkovic@isi.edu

ABSTRACT

Malware analysis uses debuggers to understand and manipulate the behaviors of stripped binaries. To circumvent analysis, malware applies a variety of anti-debugging techniques, such as self-modifying, checking for or removing breakpoints, hijacking keyboard and mouse events, escaping the debugger, etc. Most state-of-the-art debuggers are vulnerable to these anti-debugging techniques.

In this paper, we first systematically analyze the spectrum of possible anti-debugging techniques and compile a list of 79 attack vectors. We then propose a framework, called *Apate*, which detects and defeats each of these attack vectors, by performing: (1) just-in-time disassembling based on single-stepping, (2) careful monitoring of the debuggee's execution and, when needed, modification of the debuggee's states to hide the debugger's presence. We implement Apate as an extension to WinDbg and extensively evaluate it using five different datasets, with known and new malware samples. Apate outperforms other debugger-hiding technologies by a wide margin, addressing 58%–465% more attack vectors.

CCS Concepts

•**Security and privacy** → **Software reverse engineering;** *Malware and mitigation;* Software security engineering;

Keywords

Malware Analysis, Anti-Debugging

*This project is the result of funding provided by the Science and Technology Directorate of the United States Department of Homeland Security under contract number HSHQDC-16-C-00024. The views and conclusions contained herein are those of the authors only.

SAC 2017,April 03 - 07, 2017, Marrakech, Morocco

http://dx.doi.org/10.1145/3019612.3019791

1. INTRODUCTION

Debuggers enable detailed analysis of malware's behaviors, including disassembling of the binary code, capturing of the system calls and the exceptions, etc. Malware authors have strong incentives to make this analysis as difficult as possible. Malicious binaries exhibit *evasive behaviors* [4, 5, 13], which aim to detect or disrupt the analysis in VMs or in debuggers. Many contemporary malware samples use evasive behaviors. Chen et al. [5] find that 39.9% and 2.7% of 6,222 malware samples exhibit anti-debugging and anti-virtualization behaviors respectively. Branco et al. [4] find that 43.21% of 4 M samples exhibit certain anti-debugging behaviors and 81.4% exhibit anti-VM behaviors. In **anti-debugging**, malware detects debuggers by searching for artifacts used to implement core debugger functionalities, such as breakpoints or tracing [8, 20]. Popular debuggers today, such as IDA [10], WinDbg [25], and OllyDbg [26], are all vulnerable to anti-debugging. There are extensions to these debuggers, which can disclose some attack vectors but not the others. Similarly, research approaches against anti-debugging (Sect. 5) cover a small subset of attack vectors.

Contributions. We propose Apate – a framework for systematic debugger hiding. Our **first** contribution lies in the systematic investigation of known and possible anti-debugging attack vectors from a variety of sources [4, 5, 8, 20, 27]. Our final set contains 79 attack vectors, 12 of which are identified by us. We abstract the 79 attack vectors into 6 broad categories (17 subcategories), which enables us to devise defense approaches per category. **Second**, we develop original techniques for handling attacks in two out of our six categories (suppressible exceptions and local timing), and refine and evaluate ideas sketched by prior work for three other categories. Our debugger-hiding approaches jointly handle all 79 attack vectors, while commercial debuggers and research solutions handle only 22%∼67% of those attack vectors [4, 6, 8, 10, 15, 18–20, 26]. **Third**, we have implemented our debugger-hiding approaches in Apate as an extension to the popular debugger WinDbg. It augments the existing debuggers with debugger-hiding capabilities and is debugger-agnostic and OS-agnostic. While the implementations of some specific attack vectors and defense mechanisms depend on our platform (Windows with WinDbg), the basic attack and defense strategies are portable to other OS and debuggers. In addition, our systematic way of handling attacks and the extensible architecture of Apate, make it a promising tool to handle future malware evasion.

All source code and evaluation materials used in our work are available at: `http://steel.isi.edu/Projects/apate/`.

2. ATTACK VECTORS

In this section, we categorize attack vectors that malware can use to detect and evade debuggers. At the top level, we differentiate between attacks that build on debugging principles and attacks that leverage traces left by debuggers. In our evaluation, popular debuggers could address some but not all the attacks we discuss in this Section.

2.1 Attacks On Debugging Principles

Attacks in this category exploit the interactions between a debugger and its debuggee in an active debugging session. They detect mechanisms employed by debuggers for code analysis, such as breakpoints, exception handling, code disassembly, etc. The challenge in handling these attacks is that core debugger functionalities must be preserved.

Breakpoint Attacks. Breakpoint attacks seek to detect and/or evade breakpoints, which are the debugging mechanisms used to closely examine the debuggee's behavior. This category contains three subcategories: **software read**, **software write** and **hardware read**.

To add a software breakpoint in a debuggee, a debugger replaces the opcode byte at the breakpoint address with a `0xcc` byte (disassembled as an `int 3` instruction). When this instruction is executed, a breakpoint exception will be raised and passed to the debugger by Windows. To add a hardware breakpoint, a debugger saves the breakpoint address in a debug register rather than modifying a debuggee's code. Both software and hardware breakpoints can be detected or evaded by malware. To discover a software breakpoint, malware may scan its code for `0xcc` byte or it could evade by overwriting its code. These attacks fall into **software read** and **software write** subcategories in Table 1. To detect a hardware breakpoint (subcategory **hardware read** in Table 1), malware can read the debug registers in CPU.

Exception Attacks. Exception attacks leverage the way exceptions are handled by Windows in the presence of a debugger, to detect or evade debugging. This category contains the following subcategories: **suppressible exception**, **non-suppressible exception**, and **special-case** attacks. We notice that suppressible exception attacks have not been previously discussed in literature.

Single-stepping is one of the key mechanisms used by debuggers to step through the debuggee code. It is implemented by setting the trap flag, which raises a single-stepping exception after the next instruction is executed.

Malware can misuse the Windows's exception handling mechanism to detect debuggers. In **suppressible exception** attacks, malware raises an exception, which Windows does not pass to applications. Windows will, however, pass such exceptions to the debugger, which may pass them to debuggee during exception handling. Malware registers a custom handler for these suppressible exceptions and detects presence of a debugger if the handler is invoked. Conversely, **non-suppressible** exceptions are always passed to applications by Windows. The presence of a debugger can be detected if a non-suppressible exception is consumed by the debugger. There are also certain **special cases** of exception attacks, which require us to perform additional handling (Section 3).

Flow Control Attacks. Attacks in this category abuse the implicit flow control mechanism that is available in the Windows operating systems, with the goal of executing out-of-debugger. This category includes **callback**, **direct hiding**, **multi-threading**, and **self-debugging** subcategories.

The implicit flow control is typically implemented through *callbacks*, such as `CallMaster()`, enumeration functions, thread local storage (TLS), and many others. These callbacks, usually take a function address as a parameter [8]. When a debugger steps over a callback, the execution flow will be transferred to the function specified as its parameter. Malware exploits this in a **callback attack**, by registering a callback function and performing its malicious activities there, unseen by the debugger. Callback attacks using some APIs have been previously discussed in literature, but we discover eight new APIs that can be misused for these attacks (shown in blue text in Table 1).

In **direct hiding** attacks, malware calls certain system APIs to decouple itself from a debugger. In **multi-threading** attacks, malware hides malicious behaviors by launching different threads which run outside of the debugger. In **self-debugging**, malware spawns a child process which attempts to debug its parent. Because any given process can only be debugged by one debugger, the child process will fail, revealing the presence of the debugger.

Interaction Attacks. In this category of attacks, malware interferes with communication channels between a user and a debugger, or it attempts to detect a debugger by slow execution. This category includes **hijacking** and **timing** attacks. In **hijacking** attacks, malware uses system APIs to hijack a defender's mouse, keyboard, or screen. Once successful, the effect will remain until the malware process exits. In **timing** attacks, malware aims to detect substantial time delays introduced by interactive debugging. Malware can either query local time (via system APIs), or network time (via external time sources).

2.2 Detecting Debugger Traces

In addition to interfering with or analyzing the debugger's execution, malware can attempt to detect or circumvent debuggers by looking for traces of their presence in the file system and memory. Malware can read the file system or memory directly (**direct-read** subcategory) or via APIs (**indirect-read** subcategory). In **direct read** attacks, malware looks for debugger traces in memory and registers using assembly code. While some direct read attack vectors have been discovered before, we discover two new ones (shown in blue text in Table 1). In **indirect read** attacks, malware calls Windows APIs to detect a debugger. Some of these APIs are designed for debugger detection; others are designed for different purposes but can be re-purposed to detect debuggers. For example, when malware calls `IsDebuggerPresent()`, it will return a non-zero value if the Process Environment Block contains the field `BeingDebugged`.

Table 1: Classification of attack vectors, and their handling in Apate

Category	Sub-category	Representative Attacks	Apate's Handling	Handling Novelty
Breakpoints	Software read	`0xcc` scan	Keep a copy and feed the original byte	This is partly sketched by [8] but generalized, implemented and tested by us
	Software write	`WriteProcessMemory()`, `mov`	Update Apate's copy	
	Hardware read	N/A in Apate	N/A in Apate	
Exception	Suppressible	`EXCEPTION_INVALID_HANDLE`, `EXCEPTION_HANDLE_NOT_CLOSABLE`	Consume the exception	This is a new handling approach proposed by us
	Non-suppress.	all other exceptions	1. Monitor handler installation and add bpt at entry; 2. Pass exception to malware; 3. Monitor handler's completion and add bpt at resume address	
	Special cases	Single-stepping (Clear the trap flag to disable it)	Raise the exception	This is partly sketched by [8] but generalized, implemented and tested by us
		`int 2d` (Debuggers use a different resume address than native run)	Use correct exception resume address	
		`int 3` (Debuggee intentionally raises a software breakpoint exception)	Modify our single-stepping exception to fake it and pass to debuggee	
Flow control	Callbacks	`CallMaster()`, TLS, `MouseProc()`, `EnumDateFormats()`, `EnumDateFormatsEx()`, `EnumSystemLocale()`, `EnumSystemCodePages()`, `EnumSystemLanguageGroups()`, `EnumSystemGeoID()`, `EnumTimeFormats()`	Add breakpoints at the entry points of the callback functions	This is a new handling approach proposed by us
	Direct hiding	`ZwSetInformationThread()`, `NtSetInformationThread()`	Skip the APIs	This handling is proposed and implemented by [18, 19]. We do the same in Apate
	Multi-threading	`CreateThread()`	Set breakpoints at entries	
	Self-debugging	Child process debugs the parent process	Set `DebugPort` to `0`	
Interaction	Hijacking	`BlockInput()`, `SwitchDesktop()`†	Skip the APIs	
	Local Timing	`GetLocalTime()`, `GetTickCount()`, `KiGetTickCount()`, `timeGetTime()`, `QueryPerformanceCounter()`, `rdtsc`	Maintain a high-fidelity time source	This is a new handling approach proposed by us
	Network Timing	Query external time source, possibly via encrypted conn.	N/A	Still an open research problem
Anti-disassembly	Inst. overlap	Embed one instruction in another	Single-stepping/Tracing	This is partly sketched by [8] but generalized, implemented and tested by us
	Self-modifying	`xor` code or copy from data section		
Traces	Indirect read	`int 2e`, `NtQueryInformationProcess(DebugObjectHandle)`	Modify debuggee states after calling/skipping these APIs	
	Direct read	`ProcessHeapFlags`, `ProcessHeapForceFlags` `BeingDebugged`, `Heap`, `NtGlobalFlag`,	Overwrite with correct values when launching client	
		segment selector registers (`gs`, `fs`, `cs`, `ds`)†; `eflags` manipulation (`popf`/`popfd`/`pop ss`)†	Maintain a copy and feed the original value	This is a new handling approach proposed by us

† Patterns consisting of multiple instructions/API calls

2.3 Completeness

We aimed to be as comprehensive as possible when enumerating possible anti-debugging techniques. Starting from our sixteen sub-categories and the Windows API manual [2], we have identified 79 possible attack vectors. We cannot prove that this list is complete, but we believe it comes close *for documented Windows OS and WinDbg functionalities*, for the following reasons. First, the attacks that use Windows APIs only require enumeration of these APIs to be complete, which we have done using the Windows API manual [2]. Second, attacks that read or write system registers (e.g., `eflags`) can only use a few of Intel x86 commands, comprehensively described in the Intel's x86 manual [11], which we used. Finally, the traces left by the WinDbg are well understood and documented in [25]. This leaves a few possible sources of incompleteness, which we discuss in Section 6.

3. APATE

Apate is a collection of debugger-hiding techniques, which systematically defeat our 79 attack vectors to hide a debugger from malware. In this Section we describe the operation of Apate as it is integrated with a debugger. We designed Apate to be as platform-independent as possible, although 5 out of 17 handler implementations are specific to the Windows OS. When porting to another OS, these 5 handlers would have to be reimplemented.

3.1 Overview

Figure 1 shows the general operation of Apate. We first pre-process the debuggee by parsing its portable executable (PE) header [14]. From the header, we extract the entry point and TLS callbacks (if any). We then add software breakpoints at these locations. The user-space code of the debuggee will start from one of these locations, which allows

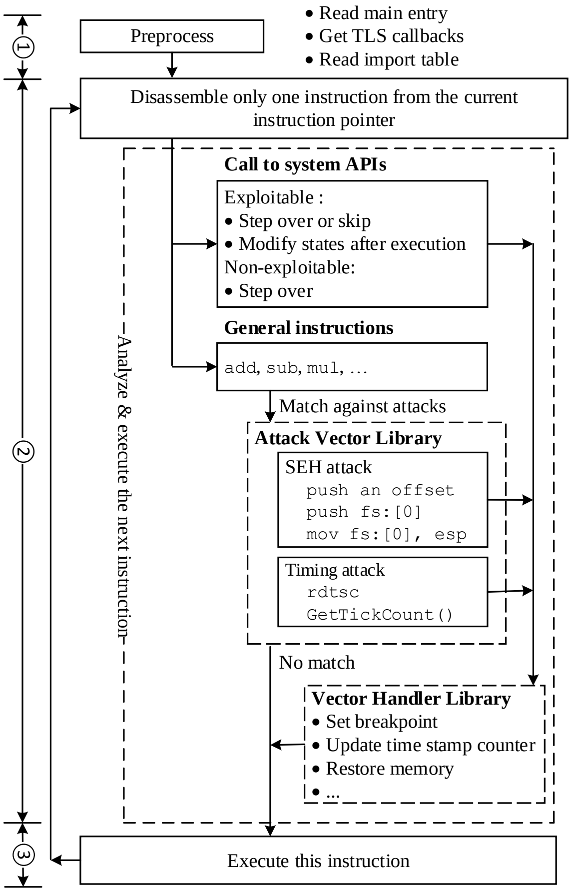

Figure 1: Overview of Apate's Operation

Apate to get control of the program from the beginning.

Next, we start from the entry point discovered in the preprocess stage, and *single-step* through each instruction in the debuggee. This single-stepping helps us thwart anti-disassembly attacks, which is achieved by setting the trap flag in the `eflags` register. The cost of single-stepping lies in the additional time it takes to analyze malware. Apate is 2.4–2.8× slower than other debuggers (Table 5), but single-stepping greatly aids its detection of anti-debugging. Apate detects between 58% and 465% more attack vectors than other debugger-hiding approaches (Section 4).

When Apate receives its first chance to handle the single-stepping exception, we disassemble and analyze the instruction that is about to be executed. Based on the instruction's semantics, we make a decision on its handling policy. We have classified all the instructions in the Intel x86 instruction set as either *calls* (conditional and unconditional jumps) or *general instructions* (everything else). If an instruction is a `call`, we check if its destination resides in the user or the kernel space. A user-space destination means that the debuggee is calling its sub-routines, so Apate single-steps into the call, which allows the defenders to analyze the entire set of malware functionalities. If the call is invoking a system API, Apate checks whether this API is in its list of possibly exploitable APIs and may step over it or skip it (see Section 3.2). For a general instruction (e.g., `add`), Apate single-steps it as it is, unless it is part of one of our 79 attack vectors. If this is the case, a vector-specific handling will be invoked. Finally, we may need to modify the debuggee state after executing the instruction to hide the Apate's presence.

While we believe we were comprehensive in our enumeration of attack vectors known to date (plus 12 new vectors discovered by us), future malware attacks may devise new vectors. New vectors can be easily added to Apate's attack vector library, and new handlers for these vectors can be added to Apate's vector handler library.

3.2 Handling Anti-Debugging

In Section 2, we discussed 17 subcategories of attacks that aim to either detect or evade debuggers. In this section, we illustrate how we handle 15 of these attacks in Apate (we skip hardware breakpoints as Apate only uses software breakpoints). This is also summarized in the 4th column in Table 1. Out of 15 subcategories, we propose novel handlers for four. For another eight subcategories, prior literature [8] has sketched some ideas for possible defenses, but has not explored all attack vectors nor described a generic solution to a given attack category. For the remaining three attack-vector subcategories, we borrow handlers proposed and implemented by others. The 5th column in Table 1 summarizes our novel contributions to attack handling.

Breakpoint attacks. Apate only uses software breakpoints, which replace the opcode of the instruction with a `0xcc` byte. This byte will raise an exception upon execution, and Windows will first allow Apate to handle this exception. To thwart **software read** and **software write** attacks, Apate performs several actions. First, when a software breakpoint is set, Apate records the breakpoint address in a lookup table called *breakpoint table*, along with the original opcode. Second, during an active debugging session, it monitors the debuggee's access to its code section and compares the target of each read and write instruction against the contents of the address field in the breakpoint table. On a match from a read instruction, Apate returns the original opcode from the table; whereas in a write, Apate updates the opcode value in the table. Our handling of breakpoint attacks has been sketched in [8], but the authors have not generalized nor implemented this countermeasure.

An interesting case occurs when the malware's instruction at the breakpoint address is already an `int 3`. This scenario requires special handling, which is also our novel contribution. We discuss this handling in Section 3.3.

Exception Attacks. Handling exception attacks is challenging. Malware can exploit a structured exception handler (SEH), a vectored exception handler (VEH), or an unhandled exception filter (UEF) to set up exception handlers. After the debuggee sets a handler, it can raise exceptions explicitly (e.g., `int`) or implicitly (e.g., divide by `0`). When handling exceptions, Apate passes **non-suppressible** exceptions to the debuggee but consumes **suppressible** exceptions. Before passing the non-suppressible exceptions, Apate also sets a breakpoint at the handler entry.

When an exception handler completes, Windows will direct the debuggee to the return address saved in its exception record. Malware may tamper with the return address. To handle this, Apate records the location of the return address on the stack during its first chance of handling the exception. Apate steps into all debuggee's exception handlers and single-steps through their instructions. When a `ret` is encountered, Apate fetches the return address from the stored location which may have been modified by malware, and sets a breakpoint at that location. This keeps malware under Apate's control when they attempt to escape.

To our best knowledge, none of the current research works differentiate between suppressible and non-suppressible exceptions. In addition, Apate sets breakpoints at handler entry and fetches the return address right before the handler returns. This novel strategy enables Apate to fully analyze malware and thwart escape attempts.

Flow Control Attacks. Attacks in this subcategory must be handled carefully; otherwise, malware can escape the debugger. For **callback** and **multi-threading** attacks, Apate will insert a software breakpoint at the entry of the callbacks or at the start address of the thread. These breakpoints will transfer the control to Apate, enabling defenders to fully analyze malware. Apate will skip the execution of the APIs in **direct hiding** subcategory, by adding the size of the current instruction to `ip`. To bypass **self-debugging** check, Apate sets `EPROCESS->DebugPort` field to `0`. This allows another debugger to attach to the same process as Apate, which is similar to the handling in [22].

Interaction Attacks. Apate will skip execution of system calls in the **hijacking** subcategory. If an API changes any system state, Apate mimics this effect to create an impression of faithful execution to malware.

The **timing** attacks can be very complex, as malware uses many sources of time to detect debuggers. We handle only attacks that use internal time sources (e.g., system clock). Handling attacks that use external time sources is an open research problem [27], and we leave it for future work.

To defeat timing attacks that leverage internal sources, Apate maintains a software time counter and uses it to adjust the return values of time queries. We update our time counter by adding a small delta which reflects the CPU cycles for each malware instruction that has been executed. We also add a small, randomly chosen offset to the final value of the time counter, which can defeat attempts to detect identical timing of repeated runs. Previous works [7,23] only add a constant value to their time sources, which can be detected if malware measures whether the elapsed time is the same across runs.

Debugger Traces. To hide the debugger traces, we have enumerated memory locations and registers that may be used to store these traces, and the Windows APIs that access these locations. Apate compares each debuggee's instruction and its parameters with this list of APIs and locations, to detect **indirect read** attacks. Upon a match, Apate provides a consistent, fake reply which hides the debugger's presence.

Instead of using APIs, malware may read memory directly to look for debugger traces. To handle **direct read** attacks, Apate detects accesses to the items in our list of debugger trace locations, and overwrites contents at these locations to hide debugger's presence. These actions do not affect the accuracy of the debugger's execution. Some strategies for handling indirect and direct read attacks were mentioned by [8], but they were specific to a few attack vectors. We generalized these strategies, and implemented and tested them. We also propose and implement two novel handlers for our newly discovered attacks, which use `cs` and `ds` registers.

3.3 Attacks Against Apate

There are several possible attacks on Apate, which could lead to malware detecting its presence. We have developed special handlers for these attacks, which we describe below. These handlers are also our novel contributions.

Our Apate framework sets the trap flag each time it single-steps an instruction. If malware reads the trap flag, it can detect the debugger's presence. Similarly, malware may clear the trap flag and check it afterwards to detect a debugger. All reads and writes of the trap flag occur through a few dedicated instructions, listed in "direct read" subcategory of Table 1 (`pushf/pushfd/popf/popfd`, `pop ss`). These reads and writes are detected by Apate. We handle the attacks by creating a "debuggee-only" version of the trap flag. Malware reads and writes manipulate this copy.

If malware sets the trap flag and Apate consumes the corresponding single-stepping exception, malware can detect the debugger's presence. To handle this case correctly, Apate needs to consume the single-stepping exceptions generated by itself, but pass those raised by the debuggee. Apate detects this case by checking the presence of the value `1` in the debuggee-only version of the trap flag. If the single-stepping exception is intentionally raised by the debuggee, Apate will faithfully pass it to the debuggee.

The next attack is specific to WinDbg, which is our chosen integration platform. WinDbg engine has a special handling for the software breakpoint exception, which is intentionally raised by the debuggee (`int 3`). WinDbg will lose control when single-stepping this instruction. Since WinDbg is closed-source software, we could not diagnose the reason behind this occurrence. To work around this problem, when Apate single-steps the instruction preceding `int 3`, it will modify the single-stepping exception record on the stack to transform it into a software breakpoint record. Specifically, we change the exception code to be `EXCEPTION_BREAKPOINT` and also update the exception address to be the beginning of the `int 3` instruction. This enables Apate to retain control and step into the exception handler for `int 3`.

3.4 Uses of Apate

Apate can be used to automatically single-step instructions in malware binaries, and record disassembled instructions and system traces for further analysis. In this use case, Apate compares each instruction against attack vectors in its library, and applies countermeasures automatically where needed. This slows down the analysis (up to 2.8× in our tests), but it may be acceptable, as malware analysis is frequently performed in an automated, batch fashion.

Apate can also be used to assist interactive debugging, where the users use single-stepping only when they desire to closely examine a portion of malware code. In this case, the overhead introduced by Apate's single-stepping is negligible, compared to user think time.

4. EVALUATION

In this section, we compare Apate against several mainstream debuggers, using five data sets (Table 2). We performed our testing on DeterLab testbed [3]. we use Windows 7 Pro x86 with SP1 (retail build) and we integrate Apate with WinDbg v6.3 x86. The physical machine has Intel Xeon CPU E3-1245 V2 @ 3.40 GHz, with 4 GB memory, and a hard drive of 1 TB.

4.1 Anti-Debugging is Prevalent

We randomly select 1,131 binaries from Open Malware [9] that are captured from 2006 to 2015. These samples are then sent to a malware analysis website VirusTotal [21], which uses 20∼50 anti-virus products to analyze each binary. We retain the binaries detected as malicious by more than 50% anti-virus products. This leaves us with 881 samples. Each binary is automatically single-stepped for a maximum of 20 minutes under Apate. Some works [17] run samples for up to several hours; however, their goal is to explore all execution paths, while our goal is to detect the anti-debugging checks, which usually occur at the beginning of a run.

Spectrum of anti-debugging techniques. Table 3 displays the details of anti-debugging techniques which are detected in our samples. The third column shows the number (and, where interesting, the percentage) of samples that apply a particular anti-debugging check. The last column shows the maximum number of times the given check was used by a single sample. We highlight only a few major findings. In the "Traces" category, checking the trap flag is the most popular anti-debugging technique, used in 15% of the samples. Our results indicate that 83 samples read or write to the trap flag to detect debuggers, and one sample conducts 102,162 instances of the trap flag attack. In the APIs category, `int 2e` attack is the most popular detection technique, adopted by 2% of our samples. This instruction is actually a system call and does not raise any exceptions.

4.2 Apate Outperforms Other Debuggers

Using our enumeration of attack vectors in Table 1, we design 79 tests cases, one per each vector. Table 4 gives the number of test cases in each attack category, and the rest of the table lists the numbers of test cases handled by each debugger. Different test cases need to be evaluated in specific ways such as single-stepping, setting a breakpoint in the code, free execution, etc. We will release all test cases and evaluation scripts on our project website.

We compare Apate to several popular debuggers: WinDbg, IDA Pro, OllyDbg, and Immunity Debugger [6]. Where possible, we evaluate both a basic version of a debugger and any extensions that aim to handle anti-debugging. IDA Pro's version is 6.6, with two highly-ranked debugger-hiding plugins: Stealth v1.3.3 [15] and ScyllaHide v1.2 [19]. We evaluate OllyDbg 2.01 and two debugger-hiding plugins: OllyExt v1.8 [18] and ScyllaHide v1.2. Since the aadp v0.2.1 [1] plugin only works in OllyDbg v1, we switch to the latest v1.10 when testing aadp. Each test case takes about a few seconds to evaluate in each debugger.

Results. We find that all basic versions of the debuggers can only handle a limited number of attack vectors. WinDbg achieves the best performance, identifying 22 out of 79 vectors, while OllyDbg and IDA Pro are able to handle 21 and 17 respectively. Plugins substantially improve the debuggers' robustness. For example, IDA Pro with Stealth and ScyllaHide plugins handles $43/17 = 2.5\times$ more anti-debugging techniques than its base version. Apate can handle all 79 test cases. Compared to the second best debugger – OllyDbg with OllyExt, Apate outperforms it by $(79 - 50)/50 = 58\%$.

4.3 Apate Detects Known Vectors

In this section, we evaluate Apate using 4 malware samples (Table 5), which are known to employ heavy anti-debugging techniques and have been manually analyzed by others. We also compare the performance of Apate with OllyDbg with OllyExt extension, its closest competitor from the previous evaluation. For brevity, we denote "OllyDbg with OllyExt" as just "OllyExt". In our evaluation, we set both Apate and OllyExt to automatically single-step through the samples until they exit.

Results. Table 5 shows the results. Apate overcomes all the anti-debugging techniques in each sample, while OllyExt fails to detect between one and three checks per sample. Furthermore, Apate finds that the first sample also performs trap flag attack, which manual analysis has missed [24].

Time Cost. In our evaluation, Apate performs 2.4∼2.8× slower than OllyExt. This is expected, because Apate considers more anti-debugging checks for each instruction and handles more attack vectors, which improves the accuracy of malware analysis.

4.4 Apate Deceives Malware

In this section, we evaluate if Apate can successfully hide a debugger's presence. For chosen 20 samples, we compare a sample's functionalities in a native run (without any debugger) with its functionalities when run under Apate. If these two match, we conclude that the debugger was hidden.

Native functionality. We enumerate a malware's functionalities by recording its file and network activities. We use these as proxies for malicious behavior, as they are necessary to exfiltrate data from compromised hosts, or send new commands/data to these hosts. To filter out noise, we first observe a base OS's file and network activities, without malware, in six runs. We create a union of all the created, deleted, and modified files, and all network communications – U_{base}. Next, we perform three native malware runs and create an intersection of activities found in all three runs – I_{native}. We define the set difference $S_{sig} = I_{native} - U_{base}$ as a malware's *signature*. In our evaluation, we look for all the items from this signature to determine if malware performs the same malicious activities with and without Apate.

Table 2: Data Sets for Evaluation

Name	Num	Goal	Findings
Unknown malware	881	Find spectrum of anti-debugging techniques	1) Malware uses 0∼10 distinct anti-debugging checks; 2) A single check can be used up to 695,219 times.
Vector tests	79	Evaluate popular debuggers v.s. Apate	1) Apate addresses all the attacks; 2) The second best debugger solves 50
Known malware	4	Demonstrate practical use of Apate	Apate finds all the anti-debugging techniques in the samples
Unknown malware	20	Prove Apate effectively hides a debugger from malware	The samples show the same malicious behaviors in Apate as in a physical machine
Packed binary	10	Prove Apate outperforms other research solutiosn	Apate overcomes all the anti-debugging techniques provided by commercial packers

Table 3: Spectrum of Anti-debugging Techniques

Cat.	Details	Samples	Max
Traces	Trap flag	83/15%	102,162
	`CheckRemoteDebuggerPresent()`	10	1
APIs	`int 2e`	11/2%	1
	`CreateFileA()`	5	5
	`OutputDebugString()`	3	90
	`FindWindow('OLLYDBG')`	1	1
Soft. bp/int	`int 3`	45/8.5%	6
	`int 1`	1	1
Except.	`ACCESS_VIOLATION`	116/22%	776
	`PRIVILEGED_INSTRUCTION`	13	2
	`ILLEGAL_INSTRUCTION`	11	2
	`INTEGER_DIVIDE_BY_ZERO`	3	1
Interact.	`GetTickCount()`	141/27%	695,219
	`QueryPerformanceCounter()`	82	1
	`rdtsc`	14	216,120
	`GetLocalTime()`	8	156
	`BlockInput()`	6	1
Implicit flow	SEH	308/58%	148
	`UnhandledExceptionFilter()`	40	9
	TLS callback	16	2
	`AddVectoredExceptionHandler()`	1	278
Disassem.	Self-modifying	145/27%	3,423
	Instruction overlapping	125	805

Results. We test 20 random malware samples that make moderate use of anti-debugging techniques as found by Apate. We set both Apate and OllyExt to single-step through the samples automatically, similar to the method used in Section 4.3. Malware functionalities under Apate were identical to those in native runs for all twenty samples. We thus conclude that Apate successfully hides the debugger's presence from malware. In OllyExt runs, however, nine out of twenty malware samples show reduced activity, both with regard to file accesses and network traffic. This indicates that malware has detected OllyExt in 45% of the cases.

5. RELATED WORK

Covert Debugging. The goal of Apate is to help the defenders identify and overcome anti-debugging techniques in malware. Zhang et al. [27] present a debugging framework MALT that employs System Management Mode (SMM) of CPU to transparently study evasive malware. MALT installs the debugging functionalities in the entities provided by BIOS. The paper lists 20 out of our 79 anti-debugging techniques but it is unclear how many they can handle. Vasudevan et al. [23] propose Cobra that divides the debuggee's code into blocks based on branching instructions. To overcome anti-analysis checks, the authors take two approaches. First, Cobra scans for instructions that betray the real state of the program, and replaces them with custom functions. Second, Cobra maintains a copy of memory that mimics the system states without debugger presence, and feeds this state to malware upon queries. From its design, we conclude that Cobra does not analyze certain exceptions or system APIs for anti-debugging checks, and thus could not handle suppressible exceptions, enumeration functions, or indirect read attacks. Thus Cobra would miss approximately 50% of our attack vectors.

Virtual Machine Frameworks. Some works approach the anti-debugging problem by shifting the debugging functionalities into the virtual machine monitors, under the assumption that in-guest modules cannot detect out-of-guest systems. For example, Ether [7] develops a virtual machine based on hardware virtualization and incorporates certain debugging functions in the underlying hypervisor. However, Pek et al. [16] prove that Ether can still be detected.

Anti-debugging Techniques. Some studies discuss how to classify anti-debugging techniques but they do not provide a systematic framework such as Apate, nor do they offer handlers for anti-debugging checks. Kirat et al. [12] propose MalGene, an automated technique for extracting analysis evasion signatures. MalGene leverages a bioinformatic algorithm to locate evasive behavior in system call sequences. While they are capable of extracting evasion signatures, there is no systematic enumeration of attack vectors, and we cannot compare MalGene directly to our vectors.

6. LIMITATIONS AND FUTURE WORK

While Apate surpasses other debuggers in our tests, there are some limitations that we need to address in our future work. First, our tests prove that Apate can defeat every attack vector in our library, but it is possible that there are some combinations of vectors, or some vectors we have not discovered, which Apate will not be able to handle. If new anti-debugging checks are devised in the future, Apate's library of attack vectors and handlers can be extended accordingly. Our future work will lie in standardizing these extensions and evaluating human burden. Second, in our attack vector enumeration, we did not consider the use of undocumented APIs or undocumented system objects. To mitigate this problem, we may treat all the undocumented APIs as the malware's own functions and step into them, but this will introduce substantial overhead. Third, if malware queries network time using clear-text packets, we can intercept and modify the embeded time. However, if the packets are encrypted, it will be hard to detect the timing behavior. We plan to explore these limitations in future work.

Table 4: Attack vectors, handled by different debuggers

Category	Test Cases	IDA Pro	IDA Pro/ Stealth	IDA Pro/ ScyllaHide	Olly-Dbg	OllyDbg/ OllyExt	OllyDbg/ ScyllaHide	OllyDbg/ aadp	Imm. Dbg	Win-Dbg	Apate
Traces	9	0	5	5	0	5	5	3	0	4	9
APIs	21	4	15	16	5	18	18	8	4	7	21
Hard. bp	1	0	1	1	0	1	0	0	0	0	1
Soft. bp	5	0	0	0	2	2	2	1	0	0	5
Except.	18	8	14	13	8	13	13	2	9	5	18
Interact.	9	0	2	2	0	4	3	3	0	0	9
Imp. flow	14	5	6	6	5	6	6	5	4	5	14
Disassem.	2	0	0	0	1	1	1	0	0	1	2
Total	79	17	43	43	21	50	48	22	17	22	79
		22%	54%	54%	27%	63%	61%	28%	22%	28%	100%

Table 5: Results of Known Malware Samples

No.	Apate	OllyExt	Anti-debugging Techniques	OllyExt's Failure Points
1	176 min	63 min	SEH attack, anti-disassembly, `int 2e` attack, `int 3` attack, trap flag attack (reading), self-modifying	Anti-disassembly, `int 2e` attack, trap flag attack
2	71 min	29 min	`BeingDebugged` flag (XP, Win7), `ProcessHeap` flag (XP), `NtGlobalFlag` (XP, Win7)	`ProcessHeap` flag (XP)
3	76 min	32 min	TLS callback, `FindWindow()`, `OutputDebugStringA()`	`OutputDebugStringA()`
4	65 min	26 min	`QueryPerformanceCounter()`, Exception attack, `GetTickCount()`, `rdtsc`	Exception handler entry and return address, `rdtsc`

7. REFERENCES

[1] aadp. Anti-Anti-Debugger Plugins. https://code.google.com/p/aadp/.

[2] W. API. Windows API Index. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516%28v=vs.85%29.aspx.

[3] R. Bajcsy, T. Benzel, Bishop, et al. Cyber Defense Technology Networking and Evaluation. *Commun. ACM*, 47(3), 2004.

[4] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat*, 2012.

[5] X. Chen, J. Andersen, Z. Mao, et al. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *DSN*, 2008.

[6] I. Debugger. Immunity Debugger. http://debugger.immunityinc.com/.

[7] A. Dinaburg, P. Royal, et al. Ether: Malware Analysis via Hardware virtualization Extensions. In *CCS*, 2008.

[8] P. Ferrie. The "Ultimate" Anti-Debugging Reference. http://pferrie.host22.com/.

[9] I. T. Georgia. Open Malware. http://oc.gtisc.gatech.edu/.

[10] Hex-Rays. IDA: multi-processor disassembler and debugger. https://www.hex-rays.com/products/ida/.

[11] Intel. Intel 64 and IA-32 Architectures Software DeveloperâĂŹs Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[12] D. Kirat and G. Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *CCS*, 2015.

[13] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting Environment-Sensitive Malware. In *RAID*, 2011.

[14] Microsoft. Microsoft PE and COFF Specification. https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.

[15] J. Newger. IDAStealth Plugin. http://newgre.net/idastealth.

[16] G. Pék, B. Bencsáth, and L. Buttyán. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *EuroSec*, 2011.

[17] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-Force: Force-Executing Binary Programs for Security Applications. In *USENIX Security*, 2014.

[18] F. Rce. OllyExt. https://forum.tuts4you.com/files/file/715-ollyext/.

[19] ScyllaHide. ScyllaHide. https://bitbucket.org/NtQuery/scyllahide.

[20] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[21] V. Total. VirusTotal. https://www.virustotal.com/en/.

[22] J. Tully. An Anti-Reverse Engineering Guide. http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide/.

[23] A. Vasudevan and R. Yerraballi. Cobra: fine-grained Malware Analysis using Stealth Localized-executions. In *Security and Privacy*, 2006.

[24] T. Werner. Waledac's Anti-Debugging Tricks. http://www.honeynet.org/node/550, 2010.

[25] W. Windows. WinDbg. https://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx.

[26] O. Yuschuk. OllyDbg. http://www.ollydbg.de.

[27] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. Using Hardware Features for Increased Debugging Transparency. In *Security and Privacy*, May 2015.