Fighting Evasive Malware: How to Pass the Reverse Turing Test By Utilizing a VMI-Based Human Interaction Simulator

Jan Gruber, Felix C. Freiling[1]

Abstract: Sandboxes are an indispensable tool in dynamic malware analysis today. However, modern malware often employs sandbox-detection methods to exhibit non-malicious behavior within sandboxes and therefore evade automatic analysis. One category of sandbox-detection techniques are reverse Turing tests (RTTs) to determine the presence of a human operator. In order to pass these RTTs, we propose a novel approach which builds upon virtual machine introspection (VMI) to automatically reconstruct the graphical user interface, determine clickable buttons and inject human interface device events via direct control of virtualized human interface devices in a stealthy way. We extend the VMI-based open-source sandbox DRAKVUF with our approach and show that it successfully passes RTTs commonly employed by malware in the wild to detect sandboxes.

Keywords: Malware Analysis; Sandboxing; Virtual Machine Introspection; Reverse Turing Test

1 Introduction

To cope with the vast amount of new malware samples found in the wild, automated analysis approaches are required to classify and triage the bulk of theses samples. One of the standard techniques to perform this is *automated dynamic malware analysis* using sandboxes. A sandbox is an isolated but closely monitored execution environment that tracks a program's system interactions and makes the resulting analysis available to further scrutiny.

Malware sandboxes were pioneered in Windows XP using a technique called *API-hooking* to catch system calls [WHF07]. Following the successes of these automated techniques, threat actors came up with a myriad of techniques to hinder or evade analyses [GLL14]. A common strategy of malware today is to exhibit a "split personality" meaning to refrain from the execution of its malicous payload if they were able to successfully detect that it is running in a sandbox [Ba10]. The resulting arms race between attackers and defenders initiated a trend of moving monitoring capabilities gradually deeper down the system stack towards the hardware, culminating in the use of hypervisor hooks and other hardware-assisted virtualization features like *Two-Dimensional Paging* [WHH13]. This has resulted in sandboxes that are stealthy regarding ongoing monitoring [Le14].

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department Informatik, Martensstr. 3, 91058, Erlangen, Germany; jan.gruber@fau.de, felix.freiling@fau.de

doi:10.18420/sicherheit2022_03

While being stealthy, hypervisor-based analysis environments are also rather involved. Therefore, user-level sandbox approaches like the well-known and widely used open-source sandbox *Cuckoo*[2] have remained popular. But they are restricted to user-level malware [MK21] and can also be actively evaded with methods like unhooking, API hammering and the direct use of system calls as shown by the project *anticuckoo*.[3] Therefore, it is advisable to rely on introspection-based monitoring, in order to get reliable analysis results. The open source project DRAKVUF[4] provides a powerful virtualization-based agentless black-box binary analysis system, which is presently the only open-source VMI framework that provides a frontend to function as full-fledged sandbox.[5]

1.1 Reverse Turing Tests

Apart from attempting to detect any (user-level) specific changes of an execution environment, another highly effective class of sandbox detection approaches as described by Bulazel and Yener [BY17] are *reverse Turing tests* (RTTs). The name is derived from the classical Turing test, where a human attempts to distinguish between an actual person and a computer. In this case, however, the malicious computer program tries to identify the presence of a human, in order to determine which one of its split personalities to show. Thus, instead of trying to detect technical changes of the execution environment, RTTs gather evidence of a human user interacting with the system which regularly revolve around human interface device (HID) events.

Generally, one can distinguish between three different ways in which RTTs can be performed:

(1) passively observe events related to human use (usually HID events, e.g., mouse movement and keyboard input),
(2) demand active interaction (e.g., by presenting a dialog box or generating the need to scroll down to the bottom of a window), and
(3) check signs of the usual "wear and tear" of human use in the system (e.g., clipboard content or recently opened files).

In order to remain stealthy, real-world malware commonly employs such observation techniques with respect to HID events. Since providing meaningful interactions with a running program automatically via the graphical user interface (GUI) is a difficult problem, demands for active interactions appear effective and were recently observed in the wild as well [HO20]. Therefore, we explore the utilization of VMI-based approaches to fool evasive malwares demanding user interaction.

[2] See `https://www.honeynet.org/projects/active/cuckoo-sandbox/`,
[3] See `https://github.com/therealdreg/anticuckoo`, Commit 4a1e7f2
[4] See `https://github.com/tklengyel/drakvuf`
[5] Maintained by CERT.pl, see `https://github.com/CERT-Polska/drakvuf-sandbox` and `https://drakvuf-sandbox.readthedocs.io/en/latest/`, accessed on 04.11.2021

1.2 Contributions

The paper presents a method that allows sandbox environments to pass the reverse Turing test. We designed and implemented a proof-of-concept system based on the open source binary analysis system DRAKVUF which reconstructs the GUI of an analysis guest running a Windows 7 operating system by means of virtual machine introspection to find clickable buttons.[6] The system then injects mouse clicks at those locations via the *QEMU Monitor Protocol* in a targeted manner. This is a novel approach to imitate user behavior and a first step in the direction of "meaningful interaction" with a running program, which can be conceptually transferred to other operating systems as well. We evaluate the system and show that it successfully passes all RTTs that are commonly deployed by malware in the wild.[7]

1.3 Paper Outline

We summarize the design goals of our system in Section 2. Subsequently, Section 3 describes its integration into the architecture of DRAKVUF and its building blocks as well as the inner working of our proof-of-concept to simulate human-like behaviour. In Section 4, we evaluate the efficacy of our implementation, before we summarize the paper, discuss remaining challenges, and give directions for future work in Section 5.

Due to the limited extent of this paper, we give a brief overview of anti-sandbox techniques in general and present several examples of RTTs in the appendix, which can be read independently.

2 Design Goals

Our goal was to build a system that is able to simulate human interaction and thereby counter reverse Turing tests, in order to support reliable sandbox-based dynamic analyses with low false negative detection rate.

One approach of commercial sandbox suppliers is to enable the user to remotely control the sandbox much like she would be sitting in front of a real machine after submitting her program for analysis.[8] This approach, however, just shifts the RTT to another level without solving the underlying problem. Although it leads to passing the RTTs reliably, it does not scale at all. Since there is a need to address this issue, the well-known open-source sandbox

[6] Our implementation has been published under the terms of version 2 of the GNU General Public License at `https://github.com/tklengyel/drakvuf/tree/master/src/plugins/hidsim`, Commit bc7708b

[7] The RTTs which implemented for the evaluation of the system have been added to the open-source test framework *pafish* under the terms of version 3 of the GPL and can be found at `https://github.com/a0rtega/pafish/pull/72`, Commit f68d74f

[8] Refer to the online sandbox `https://any.run/` for example

Cuckoo[9] employs countermeasures to fool the malware in regard to passive as well as active RTTs. This is accomplished by running a Python agent inside the guest VM that moves the cursor to a random location every second and clicks the left mouse button. Furthermore, it scans for windows and dialogs to automatically simulate the button clicking [Ra13] by using API-functions, like `SendMessageW` and `EnumWindows` exposed by `user32.dll`, which is accessed via Python's ctypes-interface.[10] However, as mentioned above, agent-based approaches like *Cuckoo* are inherently limited.

Our aim was to bring the capabilities to counter RTTs into a sandbox based on hypervisor hooks. We chose the black-box malware analysis system DRAKVUF as the basis for our system. Until now, its engine, which follows the paradigm of being absolutely stealthy [Le14], did neither accomodate HID event emulation nor a mechanism to provide some kind of meaningful interaction with the programs under test.

The upmost design goal for our solution was to stay absolutely transparent, following DRAKVUF's overall maxime that no analysis-related code, regardless of kernel-level or user-level, should run inside the analysis guest. At the same time, we aim for enhancing the sandbox's capabilities by a simulator of human behaviour to trick the malware into revealing its malicious intent.

Based on the above design choices, a HID-emulation module had to be built and integrated, which specifically sought to achieve the following tasks:

(a) Inject randomized HID events to pass passive RTTs,
(b) detect GUI updates and reconstruct GUI elements via VMI, and
(c) interact with dialogs.

3 Implementation

We now describe the most important aspects of the implementation of our prototype.

3.1 Integrating Into the DRAKVUF Architecture

DRAKVUF is built on top of the Xen virtual machine monitor. It runs in the control domain (*dom0*) and thereby makes use of direct memory access (DMA) through the *LibVMI* library, which was pioneered by B.D. Payne [Pa12], to gain access to the state of the fully virtualized Xen HVM (hardware virtual machine) guests, which are observed from the aforementioned privileged domain [Le14, p. 387], [LS20, p. 2].

[9] See `https://www.honeynet.org/projects/active/cuckoo-sandbox/`,

[10] See `https://github.com/cuckoosandbox/cuckoo/blob/master/cuckoo/data/analyzer/windows/modules/auxiliary/human.py`, Commit b26f88b

Within the control domain, DRAKVUF utilizes hypervisor features to control virtualization extensions provided by the CPU, such as the *Extended Page Tables* (EPT). While it relies on several techniques to transfer control to the hypervisor (VMEXIT), the utilization of breakpoint injection is the commonly used way to trap the execution. In order to do this, the #BP instruction – INT3 with instruction opcode `0xCC` – is written into the guest's memory and the CPU is configured to issue a VMEXIT, when breakpoints are executed. Xen, in turn, is setup to forward those events to the control domain, where DRAKVUF is running, which handles those events with previously configured callbacks [Le14, p. 388]. For the case of Windows guest systems, the code locations to inject INT3-instructions into, are determined by the utilization of debug symbols provided by Microsoft, which are converted with the memory forensics tool *Volatility* into a textual representation, in order to reliably infer the physical addresses of various kernel datastructures and establish a map of those with the help of the Kernel Processor Control Region at runtime [Le14, p. 389].

Another important capability utilized by DRAKVUF is Xen's alt2pm feature, which allows to create multiple EPTs and to switch between those at runtime [LS20, p. 3]. By restricting EPT access permissions of pages, it is possible to stealthily trace memory accesses, because EPT permission violations trap into the hypervisor, where the pages, which have been manipulated by breakpoint injection, could be switched transparently and thus remain hidden from Window's patchguard or a malware scanning for INT3-instructions as well as other modifications. This means, that the ongoing observation is not detectable from the perspective of the monitored virtual machine [Le16].

In addition to this core functionality, DRAKVUF houses a plugin architecture, which allows for flexible extension by the capability of installing traps and event handlers per plugin. If one of the expected events occurs in the VM, control is transferred to DRAKVUF running in the control domain, where the list of handlers for this type of event is traversed, in order to execute each subscribed handler [Ko18, pp. 112 f.].

3.2 The Human Interaction Simulator's Architecture

To implement human-like simulation capabilities, we built upon DRAKVUF's plugin architecture and introduced a component called *hidsim*, whose lifecycle is controlled by DRAKVUF's "main loop". The component is multi-threaded to a) inject HID events and b) install an event handler to parse the memory on updates of the user interface, in order to identify and extract GUI-related datastructures, as Figure 1 illustrates.

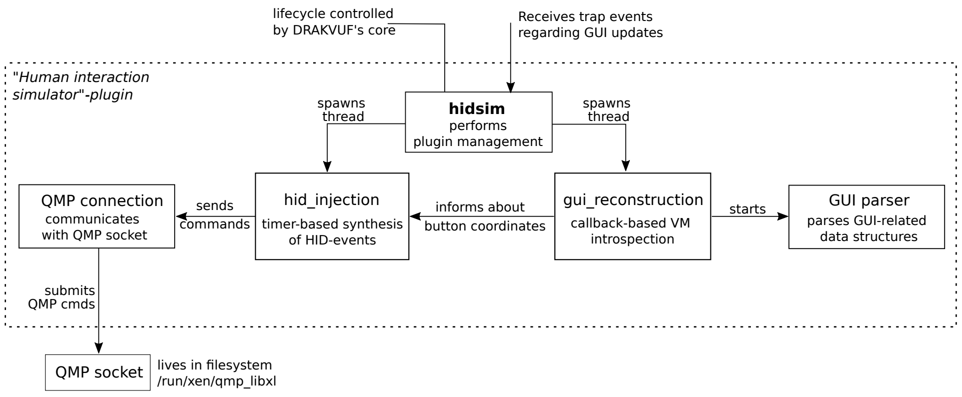
lifecycle controlled by DRAKVUF's core
Receives trap events regarding GUI updates
"Human interaction simulator"-plugin
spawns thread
hidsim performs plugin management
spawns thread
QMP connection communicates with QMP socket
sends commands
hid_injection timer-based synthesis of HID-events
informs about button coordinates
gui_reconstruction callback-based VM introspection
starts
GUI parser parses GUI-related data structures
submits QMP cmds
QMP socket
lives in filesystem /run/xen/qmp_libxl

Fig. 1: High-level overview of the implemented plugin.

3.2.1 Injection of Human Interface Device Events

Since Xen uses the QEMU device model, we could rely on the QEMU monitor protocol (QMP)[11], which allows applications to control a QEMU instance in a fine grained way by sending JSON-commands, defined by the so-called QAPI, directly to its Unix domain socket. Several of those commands allow the sending of HID events over a QMP-socket to a QEMU instance, so that its handlers update the virtualized human interface devices accordingly. From the analysis guest's perspective, this is an approach which is fully transparent. By doing so, the HID injection subcomponent is able to send either pre-recorded or randomized HID events, like cursor updates or keystrokes, following a timer-based approach. However, in order to achieve the indistinguishable illusion of a person sitting in front of the sandbox's alleged screen, the actual coordinates to click on have to be provided by a sub-component which is responsible for the reconstruction of the GUI.

3.2.2 Reconstruction of the Graphical User Interface

At the start of the analysis of a Windows 7-guest, we set up a trap on the system call `NtUserShowWindow`, which is exposed by `win32k.sys` – the kernel-mode portion of the Windows GUI subsystem [RSI12, p. 50]. We accomplish this by locating the system call-handler with the help of the GUI system service descriptor table,[12] which contains the array of pointers for each system call-handler related to USER and GDI services implemented in `win32k.sys` [RSI12, pp. 137 and 273]. Then, we insert a breakpoint, so that the plugin gets notified about updates of the GUI without employing some kind of busy-waiting, so that a

[11] See `https://wiki.qemu.org/Documentation/QMP`, accessed on 28.10.2021
[12] Also called *W32pServiceTable*

second thread can start to reconstruct the GUI based on the in-memory data structures with the help of *LibVMI* and scan for clickable buttons.

This is done by identifying and iterating all interactive window stations and their desktops, in order to retrieve the active instance of the session. Afterwards, all windows, which are represented as `tagWND`-structs in memory to be specific, are traversed to build up a depth-ordered list, which is done by following an algorithm pioneered by B. Dolan-Gavitt [Li12a].[13] Since buttons are represented as `tagWND`-structs as well, potentially clickable buttons of interest can then be determined by looking at the top *n* list elements and employing several heuristics regarding size, class ID, visibility and textual content of the `tagWnd`-struct in question. After having identified the top-most structure which fulfills the requirements, the visible part of the area and its centroid are determined and passed to the injection thread, where a corresponding HID command will be constructed and sent to the guest via QEMU's monitor protocol, so that the cursors will be moved smoothly to the given coordinates and click on those. For implementation details, the interested reader might refer to the publicly available source code.[14]

4 Evaluation

In order to evaluate the efficacy of our approach, we implemented test cases using the Win32-API in C which perform several RTTs, which were briefly addressed in Section 1.1 and will be discussed in depth in Appendix A. With the aim to contribute to the open-source community, we decided to extend the popular open-source framework *pafish*[15] by those tailored but nevertheless realistic tests. *Pafish* bundles a collection of several sandbox detection techniques used by malware in the wild [Hu15] and has already been taken up in other academic work [Yo16] but lacked most RTTs until now. The newly added implementations were published as a patch and have been incorporated into *pafish*.[16] We decided to use these handcrafted implementations, which were derived from checks found in real world malware samples, as presented in Appendix A at detail, instead of using evasive samples themselves for practical reasons. By doing so, we gain quantifiability as well as reproducibility. This is especially notable, because RTTs are often found in loaders, whose sole task is to download the actual payload from malware distribution sites, which are usually only available for a few days and would make it hard to determine, whether the RTT was passed or not.

We compared our implementation[17] with the popular open-source sandbox *Cuckoo* in version 2.0.7 with its before mentioned auxiliary module called `human.py` using VirtualBox

[13] See the originally implementation of Brendan Dolan-Gavitt in Volatility's `screenshot`-plugin

[14] See especially `https://github.com/tklengyel/drakvuf/tree/master/src/plugins/hidsim/gui`, commit 63fdbf6

[15] See `https://github.com/a0rtega/pafish`, building upon commit 62dad68.

[16] Refer to `https://github.com/a0rtega/pafish/pull/72` for our patched version of *pafish* which has been integrated into version 0.6 in the meantime

[17] Using `https://github.com/tklengyel/drakvuf/tree/master/src/plugins/hidsim`, commit bc7708b

machinery by submitting the enhanced *pafish* executable for analysis. The generated log files, where all failed tests are recorded, were extracted after each submission, in order to determine the RTTs which failed. Both sandboxes run their analyses on 32-bit Windows 7 SP1-guests, which are still common for the use case of malware analysis. We performed the analysis for exactly 60 seconds and used both systems with and without their respective plugin that is responsible for the simulation of human interaction, in order to establish causality between the tooling and the resulting observations. By performing three runs each, in which we did not observe any ambiguities, we are certain that we have eliminated fluctuations and inconsistencies. The results of this practical evaluation are depicted in Table 1.

Tab. 1: Results of running the RTTs defined in the extended *pafish* in Cuckoo and DRAKVUF, both with and without their respective human simulator plugin; The test cases correlate to the functions implemented in `rtt.c` of our patched version of *pafish*.

Sandbox	**Mouse Presence**	**Mouse Movement**	**Natural Mouse Speed**	**Single Clicking**	**Double Clicking**	**Dialog Confirmation**	**Plausible Dialog Confirmation**
Cuckoo without simulation	✓	✗	✗	✗	✗	✗	✗
Cuckoo with human.py	✓	✓	✗	✓	✓	✓	✗
DRAKVUF without simulation[18]	✓	✗	✗	✗	✗	✗	✗
DRAKVUF with hidsim[19]	✓	✓	✓	✓	✓	✓	✓

As a preliminary remark, we would like to mention that we did not compare analysis results, but considered only the efficacy of human behaviour simulation. For a comparison of the analysis capabilities, refer to [MK21].

Looking at the results, it is obvious that without any enhancements both *Cuckoo* and DRAKVUF fail all tests except the one for mouse presence. When the simulators are involved, the detection results change drastically. Our plugin passes all tests, whereas *Cuckoo*'s auxiliary package failed the check for the cursor's speed limit, since it randomly places the cursor on the screen. In addition, it failed the plausibility check added to the dialog confirmation, since it just uses `SendWindowMessageW(...)` to send a message of type `BM_CLICK` to the dialog, without taking the current cursor position into account. However, as a matter of fact, this is a made-up test, which has not been found in real world malware but is an obvious approach to determine simulation.

[18] Command: `drakvuf -d $DOM_ID -r "$KR_JSON" -W "$WIN32K_JSON" -i $PID -e "$REMOTE_PAFISH" -c "$REMOTE_CWD" -t $TIMEOUT -a regmon -a filetracer`

[19] Command as before with the following additional arguments: `-a hidsim --hid-monitor-gui --hid-random-clicks`

To summarize the evaluation, it is clearly shown, that our implementation passes all implemented RTTs and delivers a performance which is even more effective than *Cuckoo*'s auxiliary module, while it is not running any code inside the analysis guest – neither for cursor movement nor for button identification. Therefore, we claim that the proof-of-concept fulfills all of the design goals stated in Section 2.

5 Conclusion and Outlook

In the present paper, we reviewed reverse Turing tests (RTTs) as one class of evasion techniques commonly employed by malware. We surveyed implementations found in real world samples to motivate the need for appropriate countermeasures. Given the trend to move the sandboxes' monitoring measures deeper down the technology stack, we identified the necessity to realize the simulation of a human's presence without running any agent in the monitored system itself. In order to put this need into practice, we extended the VMI-based open-source sandbox DRAKVUF, which employs an absolutely stealthy monitoring approach, by a plugin to simulate a human's presence implementing a novel approach. We utilized virtual machine introspection to monitor the analysis guest for updates of its graphical user interface and to reconstruct the GUI-elements presented to the user, when such an update was detected. By doing so and utilizing QEMU's monitor protocol to control the virtualized human interface devices, we were able to perform meaningful interactions with the program under test to a certain extent, which was done by clicking on buttons of dialogs popping up. We evaluated our approach by extending the open-source testing tool called *pafish* by the RTTs found in malware samples and comparing the efficacy of our implementation with the popular open-source sandbox *Cuckoo*. This showed, that in regard to the simulation of human behaviour, our implementation is at least as effective as *Cuckoo*'s agent-based approach, but does not rely on runnning any code inside the analysis guest. Therefore, it is unsusceptible from this perspective.

Future work should try to enhance the meaningfulness of the simulated human behaviour, in order to pass more sophisticated active RTTs which malware might employ in the future. Another goal is to port our prototype to work with more recent operating systems, reconstruct the GUI in a richer way and tackle the challenge of being able to determine elements drawn by higher level GUI frameworks. Furthermore, a comparison between VMI-based and machine vision-based approaches in regard to efficacy and performance seems to be of interest. Obviously, passing the RTT in its generality remains an unsolved problem and will be subject of further research. However, the presented proof-of-concept can at least deal with several RTTs commonly employed by malware and trick it to reveal its harmful intent.

Acknowledgements

The practical implementation, on which this paper is based, was created in the course of a *Google Summer of Code* project, which was hosted by the *The Honeynet Project.*[20] Therefore, we would like to thank Michał Leszczyński and Adam Kliś (both from CERT.pl at the time) and DRAKVUF's creator Tamas K. Lengyel for their support and code reviews. Furthermore, we would like to thank Jonas Röckl, Tobias Latzo and the anonymous reviewers for helpful comments on the draft as well as the members of the Research and Training Group 2475 "Cybercrime and Forensic Computing" funded by *Deutsche Forschungsgemeinschaft* (DFG, German Research Foundation (grant number 393541319/GRK2475/1-2019) for fruitful discussions.

References

[Ba10] Balzarotti, D.; Cova, M.; Karlberger, C.; Kirda, E.; Kruegel, C.; Vigna, G.: Efficient Detection of Split Personalities in Malware. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010. The Internet Society, 2010, URL: https://www.ndss-symposium.org/ndss2010/efficient-detection-split-personalities-malware.

[BY17] Bulazel, A.; Yener, B.: A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium. Pp. 1–21, 2017.

[CS20] Chaffey, E. J.; Sgandurra, D.: Malware vs Anti-Malware Battle - Gotta Evade 'em All! In (Kohlhammer, J.; Angelini, M.; Bryan, C.; Gómez, R. R.; Prigent, N., eds.): 17th IEEE Symposium on Visualization for Cyber Security, VizSec 2020, Virtual Event, USA, October 28, 2020. IEEE, pp. 40–44, 2020, URL: https://doi.org/10.1109/VizSec51108.2020.00012.

[Fo19] Fois, Q.: Threat Actor "Cold River": Network Traffic Analysis and a Deep Dive on Agent Drable, tech. rep., Lastline Inc., Jan. 2019, URL: https://www.lastline.com/labsblog/threat-actor-cold-river-network-traffic-analysis-and-a-deep-dive-on-agent-drable/, visited on: 10/28/2021.

[GLL14] Gao, Y.; Lu, Z.; Luo, Y.: Survey on malware anti-analysis. In: Fifth International Conference on Intelligent Control and Information Processing. Pp. 270–275, 2014.

[HO20] Haughom, J.; Ortolani, S.: Evolution of Excel 4.0 Macro Weaponization, tech. rep., Lastline Inc., June 2020, URL: https://www.lastline.com/labsblog/evolution-of-excel-4-0-macro-weaponization/, visited on: 11/08/2021.

[20] See https://summerofcode.withgoogle.com/archive/2021/projects/5750586108018688/, accessed on 27.10.2021.

[Hu15] Hund, R.: Pafish: How to Test your Sandbox Against Virtualization Detection, 2015, URL: `https://www.vmray.com/cyber-security-blog/a-pafish-primer/`, visited on: 12/17/2020.

[Ko18] Kovalev, S. G.: Reading the contents of deleted and modified files in the virtualization based black-box binary analysis system Drakvuf. In: Proceedings of ISP RAS. Vol. 30. 5, 2018.

[Le14] Lengyel, T. K.; Maresca, S.; Payne, B. D.; Webster, G. D.; Vogl, S.; Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In (Jr., C. N. P.; Hahn, A.; Butler, K. R. B.; Sherr, M., eds.): Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014. ACM, pp. 386–395, 2014, URL: `https://doi.org/10.1145/2664243.2664252`.

[Le16] Lengyel, T.: Stealthy monitoring with Xen alt2pm, tech. rep., Xen Project, Apr. 2016, URL: `https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/`, visited on: 10/29/2021.

[Li12a] Ligh, M. H.: MoVP 4.2 Taking Screenshots from Memory Dumps, tech. rep., The Volatility Foundation, Oct. 2012, URL: `https://volatility-labs.blogspot.com/2012/10/movp-43-taking-screenshots-from-memory.html`, visited on: 10/30/2021.

[Li12b] Ligh, M. H.: What do Upclicker, Poison Ivy, Cuckoo, and Volatility Have in Common?, tech. rep., The Volatility Foundation, Dec. 2012, URL: `https://volatility-labs.blogspot.com/2012/12/what-do-upclicker-poison-ivy-cuckoo-and.html`, visited on: 11/07/2021.

[LS20] Leszczyński, M.; Stopczański, K.: A new open-source hypervisor-level malware monitoring and extraction system – current state and further challenges. Virus Bulletin 12/, 2020.

[MK21] Melvin, A. A. R.; Kathrine, G. J. W.: A Quest for Best: A Detailed Comparison Between Drakvuf-VMI-Based and Cuckoo Sandbox-Based Technique for Dynamic Malware Analysis. In: Intelligence in Big Data Technologies—Beyond the Hype. Springer, pp. 275–290, 2021.

[Pa12] Payne, B. D.: Simplifying virtual machine introspection using LibVMI./, Sept. 2012, URL: `https://www.osti.gov/biblio/1055635`.

[Ra13] Rapid7: Fooling malware like a boss with Cuckoo Sandbox, tech. rep., Rapid7, Apr. 2013, URL: `https://www.rapid7.com/blog/post/2013/04/16/fooling-malware-like-a-boss-with-cuckoo-sandbox/`, visited on: 10/29/2021.

[RSI12] Russinovich, M. E.; Solomon, D. A.; Ionescu, A.: Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7. Microsoft Press, USA, 2012, ISBN: 0735648735.

[SK12] Singh, A.; Khalid, Y.: Don't Click the Left Mouse Button: Introducing Trojan UpClicker, tech. rep., Fireeye Inc., Dec. 2012, URL: https://webcache.googleusercontent.com/search?q=cache:NeVZ4J1Y-cQJ:https://www.fireeye.com/blog/threat-research/2012/12/dont-click-the-left-mouse-button-trojan-upclicker.html+&cd=1&hl=en&ct=clnk&gl=de, visited on: 11/07/2021.

[VS14] Vashisht, S. O.; Singh, A.: Turing Test in Reverse: New Sandbox-Evasion Techniques Seek Human Interaction, tech. rep., Fireeye Inc., Nov. 2014, URL: https://www.fireeye.com/blog/threat-research/2014/06/turing-test-in-reverse-new-sandbox-evasion-techniques-seek-human-interaction.html, visited on: 10/28/2021.

[WHF07] Willems, C.; Holz, T.; Freiling, F. C.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Secur. Priv. 5/2, pp. 32–39, 2007, URL: https://doi.org/10.1109/MSP.2007.45.

[WHH13] Willems, C.; Hund, R.; Holz, T.: CXPinspector: Hypervisor-based, hardware-assisted system monitoring. Ruhr-Universitat Bochum, Tech. Rep/, p. 12, 2013.

[Yo16] Yokoyama, A.; Ishii, K.; Tanabe, R.; Papa, Y.; Yoshioka, K.; Matsumoto, T.; Kasama, T.; Inoue, D.; Brengel, M.; Backes, M.; Rossow, C.: SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In (Monrose, F.; Dacier, M.; Blanc, G.; García-Alfaro, J., eds.): Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings. Vol. 9854. Lecture Notes in Computer Science, Springer, pp. 165–187, 2016, URL: https://doi.org/10.1007/978-3-319-45719-2%5C_8.

A A Brief Survey of Reverse Turing Tests Found in Real World Malware

To address general interest, we now survey several reverse Turing tests found in the wild. The appendix merely illustrates the technical mechanisms of RTTs and is not necessary to understand the main part of the paper.

A.1 Passive Observation

A.1.1 A Test for Mouse Presence Found in a Malicious Document of *ColdRiver*

In its simplest form, an RTT might be to determine the availability of a mouse on the system. This is illustrated by Listing 1, which shows a part of a VBA-macro code extracted from a malicous document utilized by an APT actor called *ColdRiver* [Fo19]. There is a predicate, which ensures, that the base64-encoded payload is only fetched and decoded, if the check for the presence of a mouse device succeeded beforehand.

```
 Sub Document_Open()
   <snip>
   If Application.MouseAvailable Then
     Set DM = CreateObject("Microsoft.XML" & "DOM")
     Set EL = DM.createElement("t" & "mp")
     EL.DataType = "bin.bas" & "e64"
   <snip>
End Sub
```

List. 1: Passive reverse Turing test found in a VBA-macro of *ColdRiver*[21]

A.1.2 A Test for Mouse Movement Found in *IFSB/Gozi*

A slighty more elaborated but nevertheless primitive example for a passive observation test can be found in the leaked source code of the malware family *IFSB/Gozi*.

```
1574 #ifdef _WAIT_USER_INPUT
1575         do
1576         {
1577                 ULONG Seed = AvGetCursorMovement();
1578 
1579                 Sleep(100);
1580 
1581                 if (!Seed)
1582                 {
1583                         Status = ERROR_BADKEY;
1584                         continue;
1585                 }
1586 
1587                 Status = CsDecryptSection(g_CurrentModule, Seed % 9);
1588         } while(Status == ERROR_BADKEY);
1589 #else
1590         Status = CsDecryptSection(g_CurrentModule, 0);
1591 #endif
```

List. 2: Passive reverse Turing test found in leaked source code of the malware family *IFSB/Gozi*; taken from `install.c`, l. 1574 – l. 1591.

As it is depicted in Listing 2, the malware does only decrypt and execute its malicious payload, if cursor movements could be detected beforehand. This could be determined by repeatedly calling the subroutine `AvGetCursorMovement()`, where the API-function `GetCursorPos(&Point)` is used to calculate the delta to the previous result (Listing 3,

[21] The macro was shortened and formatted for readability after its extraction via `olevba` from the malware sample with MD5-hash 48320f502811645fa1f2f614bd8a385a.

ll. 204 ff.). If there is no such delta, the program will infinitely check for movement every 100 ms until termination (Listing 2, ll. 1575 ff.).

```
196 #ifdef _WAIT_USER_INPUT
197 
198 // // Returns mouse cursor position relative to previously saved coordinates. //
199 ULONG AvGetCursorMovement(VOID)
200 {
201         POINT Point;
202         ULONG Movement = 0;
203 
204         GetCursorPos(&Point);
205 
206         if (g_Point.x && g_Point.y)
207           Movement = Point.x - g_Point.y + ((Point.y - g_Point.y) << 16);
208 
209         g_Point.x = Point.x;
210         g_Point.y = Point.y;
211 
212         return(Movement);
213 }
214 
215 #endif // _WAIT_USER_INPUT
```

List. 3: Calculation of cursor movement found in leaked source code of the malware family *IFSB/Gozi*; taken from `av.c`, l. 196 – l. 217.[22]

The above mentioned code snippets show original source code as it was written by the author of *IFSB/Gozi* also known as *Ursnif*. They were taken from a leak of the malware in version 2.13.24.1 provided by vx-underground.org.[23]

A.1.3 A Test for Mouse Clicks Found in *UpClicker*

In 2012, the trojan *UpClicker*, which delivered a remote access toolkit called *Poison Ivy* at that time [Li12b], was one of the first malware samples which checked for mouse clicks. It utilized the API-function `SetWindowsHookExA` with the parameter `0xE` standing for `WH_MOUSE_LL` to catch mouse input events [SK12], as it is illustrated by the decompilation in Listing 4, l. 38. By doing so, an application-defined hook procedure is installed into a hook chain, which is called every time a new mouse input event is about to be posted into a thread input queue, so that it can effectively catch all of those.

```
36 void FUN_00401000(void)
37   <snip>
```

[22] As a side note: This seems to be implemented in a flawed but nevertheless working way, because coordinate axes were mixed up, as the following expression taken from the code illustrates: `Movement = Point.x - g_Point.y + ((Point.y - g_Point.y) << 16);`.

[23] See `https://github.com/vxunderground/MalwareSourceCode/blob/main/Leaks/Win32/Win32.Gozi.rar`, accessed 31.10.2021

```
  DAT_00406000 = SetWindowsHookExA(0xe,FUN_004010b0,(HINSTANCE)hmod,dwThreadId);
  iVar1 = GetMessageA((LPMSG)&local_e0,(HWND)0x0,0,0);
  while (iVar1 != 0) {
    TranslateMessage((MSG *)&local_e0);
    DispatchMessageA((MSG *)&local_e0);
    iVar1 = GetMessageA((LPMSG)&local_e0,(HWND)0x0,0,0);
  }
  UnhookWindowsHookEx(DAT_00406000);
                  /* WARNING: Subroutine does not return */
  FUN_0040129d(0);
}
```

List. 4: Decompiled view on the installation of the mouse hook procedure found *UpClicker*.[24]

The malware was dormant until the left mouse button was released after a click, which was determined by checking the event type of each received message and comparing it with the value `0x202` resembling (`WM_LBUTTONUP`), as it is shown in l. 10 of the installed callback-function presented in Listing 5. When this happened, the malware showed its real personality and injected its malicious payload into another process (l. 12).

```
1  void FUN_004010b0(int param_1,int param_2,undefined4 param_3)
2  {
3    <snip>
4    if (param_1 == 0) {
5      if (param_2 == 0x200) { // WM_MOUSEMOVE
6          <snip>
7      }
8      else {
9        if (param_2 != 0x201) {
10         if (param_2 == 0x202) { // WM_LBUTTONUP
11           UnhookWindowsHookEx(DAT\_00406000);
12           FUN_00401170();
13                     /* WARNING: Subroutine does not return */
14           FUN_0040129d(0);
15         }
16         goto LAB_004010f9;
17       }
18 <snip>
```

List. 5: Decompiled view of the relevant part of the LowLevelMouseProc installed by *UpClicker*, which inspects the mouse events.

Several variations of passive RTTs have been observed in the wild concerning movement speed, movement patterns, clicking intervals or scrolling [CS20]. Some malware, for example, used the API-Call `GetAsyncKeyState` to wait on multiple clicks, others would check the cursor's speed, and would not execute, if the mouse cursor traveled to quickly [VS14].

[24] The sample has the MD5-Hash `ce69dee5307d58db4e2a6fdbcbf87e9d`; The decompilation of was done by *Ghidra* in version 10

[24] The decompiled code was shortened and formatted for readability

A.2 Demand for Active Interaction

Since passive tests are relatively primitive and error-prone, malware author's came up with active reverse Turing tests, which utilize the graphical user interface to demand some kind of (inter)action from an eventually present human operator. Those tests can often be found in malicious documents sent by e-mail, whose sole purpose is to function as a first stage loader to retrieve the actual payload from a malware distribution site (MDS).

One instance of this technique is presenting a dialog, which asks the user for the approval of an alleged repair action after opening an Excel document. The according XLM-macro code is presented in Listing 6. The return value of the dialog is used as a predicate, which acts as an active reverse Turing test as line 3 illustrates. If, and only if, the user presses the button labelled "OK", the content of the following cells is executed, which ultimately leads to an obfuscated call of `urlmon`'s `UrlDownloadToFileA` in line 10, in order to retrieve the second stage payload.

```
 1  auto_open: auto_open->Macro251!$A$1
 2  SHEET: Macro251, Macrosheet
 3  CELL:A1 , =IF(ALERT("We found a problem with some content. Do you want to try to recover as
          ↪  much as we can?",1.0),,CLOSE(TRUE))
 4  CELL:A2 , =GET.WORKSPACE(1.0)
 5  CELL:A3 , =IF(ISNUMBER(SEARCH("Windows",A2)),,CLOSE(TRUE))
 6  CELL:A4 , =GET.WORKSPACE(32.0)
 7  CELL:A5 , =IF(ISNUMBER(SEARCH("Office",A4)),,CLOSE(TRUE))
 8  CELL:A6 , =GET.WINDOW(1.0)
 9  CELL:A7 , =IF(ISNUMBER(SEARCH("Fax",A6)),,CLOSE(TRUE))
10  CELL:A8 , =IF(GET.WORKSPACE(19.0),CALL("ur"&C6,"UR"&C7&"nloa"&C8&"ileA","JJCCJJ",0.0,GET.
          ↪ NOTE(D8),GET.NOTE(E8),0.0,0.0),CLOSE(TRUE))
11  CELL:A9 , =WAIT(NOW()+"00:00:05")
12  <snip>
13  CELL:C6 , None , lmon
14  CELL:C7 , None , LDow
15  CELL:C8 , None , dToF
```

List. 6: Active reverse Turing test found in XLM-macro.[25]

However, showing dialogs – like in this rather recent campaign from 2020 – is not the only instance of active Turing tests. Back in 2014, security analysts discovered an RTF-exploit, that would only trigger after scrolling to the second page of the document, which is what usually a curious human would do, but won't happen in an automated sandbox [VS14].

[25] Extracted with a tool called `xlmdeobfuscator` from the malicious document with MD5-hash 50d518246c2b61f5b427948f87a0aa24