A Game of "Cut and Mouse": Bypassing Antivirus by Simulating User Inputs

Ziya Alper Genç
University of Luxembourg
ziya.genc@uni.lu

Gabriele Lenzini
University of Luxembourg
gabriele.lenzini@uni.lu

Daniele Sgandurra
Royal Holloway, University of London
daniele.sgandurra@rhul.ac.uk

ABSTRACT

To protect their digital assets from malware attacks, most users and companies rely on anti-virus (AV) software. But AVs' protection is a full-time task and AVs are engaged in a *cat-and-mouse* game where malware, *e.g.,* through obfuscation and polymorphism, denial of service attacks and malformed packets and parameters, try to circumvent AV defences or make them crash. On the other hand, AVs react by complementing signature-based with anomaly or behavioral detection, and by using OS protection, standard code, and binary protection techniques. Further, malware counter-act, for instance by using adversarial inputs to avoid detection, *et cetera.* This paper investigates two novel moves for the malware side. The first one consists in simulating mouse events to control AVs, namely to send them mouse "clicks" to deactivate their protection. We prove that many AVs can be disabled in this way, and we call this class of attacks *Ghost Control*. The second one consists in controlling high-integrity white-listed applications, such as `Notepad`, by sending them keyboard events (such as "copy-and-paste") to perform malicious operations on behalf of the malware. We prove that the anti-ransomware protection feature of some AVs can be bypassed if we use `Notepad` as a "puppet" to rewrite the content of protected files as a ransomware would do. Playing with the words, and recalling the cat-and-mouse game, we call this class of attacks *Cut-and-Mouse*.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Antivirus, Ransomware, Evasion, Vulnerability, Simulated Inputs

ACM Reference Format:
Ziya Alper Genç, Gabriele Lenzini, and Daniele Sgandurra. 2019. A Game of "Cut and Mouse": Bypassing Antivirus by Simulating User Inputs. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3359789.3359844

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359844

1 INTRODUCTION

To protect IT assets, distinct classes of basic security practices are often provided to the end users depending on their usage scenario. For instance, home users are instructed to always update their operating system (OS) and applications; corporate administrators are required to employ some form of user training to teach users, *e.g.,* how not to click on e-mails that look suspicious; organizations are recommended to use firewalls to protect their networks from remote attackers. However, it is often the case that the first security recommendation given to all classes of users is to install an anti-virus (AV) on their devices. In fact, AVs are believed to be one of the best protection solutions, specifically against malware; AVs are installed in most user computers and companies, and are implicitly trusted by most users, and are part of the trusted computing base[1].

It goes without saying that, while AVs do offer protection, they cannot catch all malware. Not only there might be missing signatures in their database [2], by over the years malware authors have spent great effort in trying to evade AVs detection, *e.g.,* through obfuscation and polymorphism [35] or evasion [4], or by disabling or crashing the AV [17, 33]. This is the classical *cat-and-mouse* game between AVs and malware, in which the first class of attacks (*e.g.,* polymorphism) is typically mitigated by some form of anomaly or behavioral detection [8, 34, 34] while the second one (*e.g.,* evasion) is mitigated by making the AV more difficult to exploit, such as through OS protection and standard binary integrity protection techniques [1]. The battle continues on, as now malware can try and bypass AV behavioral detection using, for instance, adversarial inputs [7], and AVs will incorporate robust mechanisms to mitigate the effects of these inputs [10, 14].

In relation to this *cat-and-mouse* game, we raise two questions. (a) *Can an attacker instrument a malware to send mouse and keyboard events to AVs to deactivate their functionalities*? In particular, we wonder whether off-the-shelf AVs can be disabled by a malicious program's mimicking user inputs (through synthesized keyboard and mouse events) to turn off, or temporarily freeze, their operations, especially those aimed at protecting from malware attacks. In theory, we would expect that, to protect from spoofed inputs, AVs enforce some forms of integrity and authentication checks on inter-process communications and user access control to verify the legitimacy of the received inputs. Instead, in our experiments, we found that most of the AVs can be easily disabled by a malware simulating mouse clicks, in particular, AVs are stoppable by spoofing requests to their main graphical interface. (b) *Can we extend this class of attacks to control a trusted application like a "puppet" and instruct it to perform malicious operations on behalf of a malware itself*? To this end, we tested whether a ransomware can circumvent

[1] Most AVs require kernel-level privileges to perform some of their operations.

the *Protected Folders* feature of AVs by exploiting and controlling `Notepad`, a trusted application, to bypass the restrictions in place and encrypt files in the Protected Folders.

These two research questions are linked to a more fundamental issue about *human-to-process event authentication*. In question (a), in fact, the matter is whether AVs do authenticate inputs (supposedly coming from hardware operated by a user) as really coming from the user's operating the hardware. In question (b), the matter is less direct and concerns recognizing inputs as authentically coming from the user events that are generated by a user application. Since applications may have security clearance, source authentication becomes necessary to recognize events that instead are generated by a malicious software, not the user, controlling the application. Both are hard problems and the vulnerabilities that we illustrate in this paper show how critical are the attacks that root to a failure in this authentication process.

Problem Statement: This paper claims that the following problems exist in current malware mitigation:

- (**P**-i) Several AV programs contain a critical flaw that allows unauthorized agents to turn off their protection features. In detail, the real-time scanning service of some AVs can be disabled by malware. This will make victims exposed to several kinds of cyber threats, especially those originated from malware.
- (**P**-ii) The *Protected Folders* solution provided by AV vendors suffers from design weaknesses. In fact, a small set of whitelisted applications is granted privileges to write to protected folders. However, whitelisted applications are not protected from being misused by other applications. This trust is therefore unjustified, since a malware can perform operations on protected folders by using whitelisted applications as intermediaries. In particular, ransomware might be able to exploit some of the whitelisted applications to change the contents of files, thus to encrypt user data.

In this paper we discuss two classes of attacks that prove these problems currently exist, the first one being *Ghost Control*, aimed at crafting stealthy mouse events to disable AVs (**P**-i), while the second one is *Cut-and-Mouse*, aimed at simulating keyboard events to control trusted applications (**P**-ii).

1.1 Ethical Issues and Responsible Disclosure

This research may have ethical concerns of dual use. We therefore adhere to an ethical code of conduct and responsible disclosure [19, 23]. We do not disclose the names of the AV companies, nor publicly share any piece of software that can be used to exploit the vulnerabilities reported in this paper. We have dutifully engaged with the affected AV companies to inform them about our findings by following responsible disclosure practices.

2 BACKGROUND

In this section, we recap the essential background information to understand our attacks. We begin with explaining the ransomware mitigation in current antivirus solutions. Next, we summarize existing measures provided by Windows OS to protect processes from unauthorized modifications.

2.1 Ransomware Defense in AVs

In response to the rise of ransomware threat, AV vendors have developed dedicated ransomware detection modules that are either integrated into their products or as standalone tools. While internal mechanisms of AVs are not publicly documented, the available options in most of AV configuration interfaces suggest that these anti-ransomware components are primarily based on whitelists. Similar to the virus signature databases, these lists are maintained by AV vendors by default, though, users can also add additional applications that they trust.

The vendor of Windows OS, Microsoft, has also developed a specific anti-ransomware solution, called *Controlled Folder Access*, which has been included in Windows 10 Fall Creators Update (Version 1709) and Windows Server 2019. Ransomware Protection, integrated into Windows Defender antivirus, controls which applications have access to protected folders, a list of directories that includes system folders and default directories such as `Documents` and `Pictures`. Users can also add further directories to the protected folder list in order to extend the coverage of protection. By default, the decision of granting applications access to protected folder is made by Windows, hence Microsoft, but users can also allow specific applications to access the protected folders.

In this paper, we use the term *trusted applications* when referring to the applications that has write access to protected folders, either granted by AV vendor or added by the user.

2.2 Process Protection via Integrity Levels

Computer architecture we use today is designed to run multiple processes concurrently, that is, all running processes share the same execution environment. To protect processes from malicious alterations by other processes, Windows OS employs access control mechanisms. Mandatory Integrity Control (MIC) is one of these security features, which enables the OS to assign an Integrity Level (IL) to a process: this value indicates the privilege level of that process. MIC defines four values for IL, with the increasing privileges: *Low*, *Medium*, *High*, and *System*. When a process attempts to interact with another process, MIC checks IL of the initiator and prevents if the target has higher IL. For example, injecting code to another process using `CreateRemoteThread` or write data to the memory of another process via `WriteProcessMemory` will fail if the caller does not possess at least the same IL as the target.

Closely related to MIC, User Interface Privilege Isolation (UIPI) is another security feature of Windows, which complements MIC to prevent unauthorized process interactions. UIPI also utilizes ILs and blocks window messages flowing from a process with lower IL. For example, calls to `SendMessage` Application Programming Interface (API) would fail if the caller has a lower IL than the target. Specifically, UIPI prevents the Shatter attack that we review in §8.

3 THREAT MODEL

In the description of our attacks, we assume the system is protected using the latest generation of AVs with specific modules against ransomware, and with built-in anti-ransomware feature of the OS. We assume the attacker is able to get access to a Windows system with user privilege levels by either tricking the user into clicking on a file (*e.g.,* attached or linked in an email) or by exploiting a

vulnerability in the victim's system. Once the attacker has established a foothold into the system, it will typically drop/download a malware to perform malicious operations, however, the malware will be blocked by an AV, or in the case of ransomware, encryption of files in protected folders will be blocked by anti-ransomware protected folder feature offered by Windows or some AVs. Henceforth, the focus of this paper is on how attackers can bypass AVs and anti-ransomware protection modules, and in providing practical mitigation solutions, rather than in the problem of detecting and protecting the system from remote attacks. This threat model is sometimes referred as a *2nd stage attack*, meaning an attacker would need to have remote access to a victim's computer, or have installed a malicious application using one of the two previously outlined alternatives (or through other means).

In this threat model, we will perform two attacks, which are described in the next two sections: the first attack (*Cut-and-Mouse*) is aimed at bypassing the protected folder feature to encrypt files in protected folder, while the second one (*Ghost Control*) is aimed at disabling AVs' real time protection.

4 ENCRYPTING PROTECTED FOLDERS

In this section, we describe our attack, *Cut-and-Mouse*, which allows ransomware to evade detection by anti-ransomware solutions, which are based on protected folders, and to encrypt the victim's files. First, we investigate the root causes that leads to this attack. Next, we give the attack details, and finally propose a practical solution.

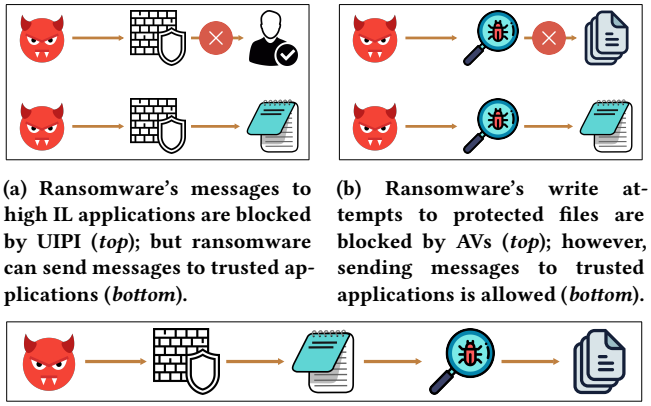
4.1 Disharmony Between UIPI and AVs

As explained in §2, anti-ransomware modules of commercial AV software grant write access to trusted applications only. To ensure this defense strategy cannot be easily bypassed, the trusted applications should be protected from any malicious modifications which would be seen in a typical malware attack. For instance, as we report the details in §6, current AVs detect when a malicious Dynamic-Link Library (DLL) module is injected into a trusted application, and suspend or kill its process. Similarly, UIPI, another protection described in §2, protects processes that run with administrative privileges from malware.

Nonetheless, we have discovered two entry points for an attack which enables malware to bypass these defense systems, namely:

- (**E**-i) *UIPI is Unaware of Trusted Applications*: UIPI filters simulated inputs based on integrity levels, however, UIPI is agnostic of the trust level assigned to applications, so it does not enforce any policy in these cases: as shown in Fig. 1a, that means that an attacker can send messages to trusted applications, in particular to those that are allowed to read and write to protected folders;
- (**E**-ii) *AVs Do Not Monitor Process Messages*: AVs do not monitor synthesized clicks or key press events flowing into the trusted applications: as depicted in Fig. 1b, this means that a ransomware can bypass protected folder enforcement by sending control messages to a trusted application.

These two entry points form a vulnerability that can enable malware to perform practical attacks, such as that shown in Fig. 1c where a ransomware can control a trusted application to perform controlled write operations as to encrypt inaccessible protected files. The attack is described in more detail in the next section.



(a) Ransomware's messages to high IL applications are blocked by UIPI (*top*); but ransomware can send messages to trusted applications (*bottom*).

(b) Ransomware's write attempts to protected files are blocked by AVs (*top*); however, sending messages to trusted applications is allowed (*bottom*).

(c) Ransomware can control a trusted application to perform write operations to protected files.

Figure 1: The disharmony between UIPI and AV software's protected folders mechanism, as described in (a) and (b), is the root cause of the vulnerability which leads to the attack depicted in (c).

4.2 Attack Overview

Using the vulnerability described in the previous section, ransomware can bypass anti-ransomware protection via controlling a trusted application and encrypt the files of the victim, including those stored in protected folders. To this end, for each file F_{target}, the ransomware performs the following tasks as depicted in Fig. 2. Firstly, ransomware reads the contents of F_{target}, which is in a protected folder (1). This is perfectly legal: in fact, reading a protected file is permitted by default[2]. The plaintext retrieved from F_{target} is encrypted in ransomware's own memory. The resulting ciphertext is then encoded in a suitable encoding format, *e.g.,* Base64 [13], and copied into the system clipboard (2). Next, the ransomware launches the `Run` window (3) to start a trusted application $App_{trusted}$, with the goal of controlling it. In this example, $App_{trusted}$ is `Notepad` as it is typically trusted in Windows environments. In addition, `Notepad` understands shortcuts for file and edit commands that ransomware will send. Using the `Run` window, ransomware executes $App_{trusted}$ with the argument F_{target}, so that the contents of F_{target} is loaded into $App_{trusted}$'s window (4). Next, the data in $App_{trusted}$'s window are selected, and overwritten with the clipboard data (the encrypted data) with a paste command (5). Finally, $App_{trusted}$ is instructed by the ransomware to save the modifications, and close the handle to F_{target} (6). All interactions in Steps 3-6 are carried out by sending keyboard inputs which are synthesized programmatically by the ransomware to control $App_{trusted}$.

The combination of these actions effectively allows ransomware to bypass the current protection methods of AVs that are aimed explicitly at blocking ransomware. Therefore, by referring to the

[2]Some AVs also provide an optional, more strict access setting that, if activated, makes AVs block the read requests from non-trusted applications.

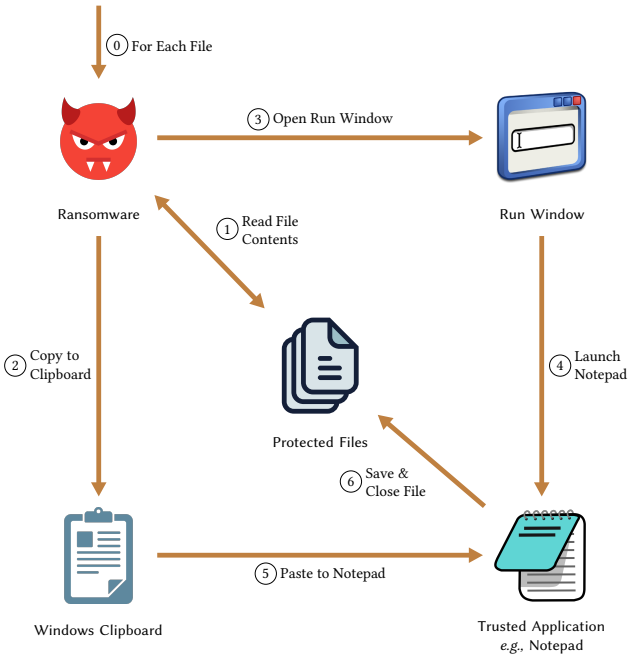


Figure 2: Bypassing anti-ransomware protection of AVs by using inputs programmatically synthesized by ransomware to control a trusted application.

never-ending 'cat-and-mouse' game of detection/anti-detection and anti-evasion/evasion among AVs and malware, and the usage of simulated keyboard and mouse inputs, we have named this attack *Cut-and-Mouse*. Algorithm 1 details the main steps of the *Cut-and-Mouse* attack.

Algorithm 1 *Cut-and-Mouse* Attack: Exploit Trusted Apps with Simulated Keyboard and Mouse Inputs to Write to Protected Folders.

```
1:  function CONTROL(App_trusted)          ▷ Application to Control.
2:      FileList ← EnumerateTargetFiles()
3:      for all f ∈ FileList do
4:          plainBytes ← f.ReadAllBytes()
5:          encBytes ← Encrypt(plainBytes)
6:          encodedText ← Base64(encBytes)
7:          CopyToClipboard(encodedText)
8:          Simulate(Run, App_trusted <f>)        ▷ Win+R
9:          Simulate(SelectAll)                   ▷ Ctrl+A
10:         Simulate(Paste)                       ▷ Ctrl+V
11:         Simulate(Save)                        ▷ Ctrl+S
12:         Simulate(Close)                       ▷ Alt+F4
13:     return Success
```

In more detail, there are two steps that are required for the *Cut-and-Mouse* attack to be successful. First, the step *Open Run Window* (3) in Fig. 2 is needed to disguise the operation of starting a trusted application as if it was executed on behalf of the user. If, instead, `Notepad` is directly executed by the ransomware, AVs would block write requests even if the rest of the attack is performed as described previously. In fact, in this example, even if `Notepad` is a trusted application (therefore allowed to write on protected folders), its parent process would be the ransomware, which is not trusted by the AVs, hence, write operations would be blocked. Secondly, as noted in Footnote 2, the step *Read File Contents* depicted in (1) in Fig. 2 can be blocked by AVs in some circumstances. For this reason, this limitation (that of not being able to read file contents) can be circumvented if ransomware exploits a trusted application to access the content on behalf of the ransomware. For example, ransomware could instruct `Notepad` to open the target file, and then synthesize two keyboard press events for `Ctrl+A` (Select All) and `Ctrl+C` (Copy), which would allow the ransomware to select all the content of the file and copy it to the system clipboard. Since the clipboard is shared between all running processes, ransomware can easily obtain the clipboard contents. It should be noted that, though, this technique might result in unrecoverable data loss with binary encoded files, due to the the presence of non-printable characters displayed by `Notepad`. However, ransomware can detect the content of the file before deciding which file to encrypt.

4.3 Mitigation Strategy

As a simple yet effective countermeasure to protect AVs modules against our *Cut-and-Mouse* attack, we suggest that trusted applications should not receive messages from non-trusted applications. That is, AVs must intercept all the messages flowing to a trusted process and block or discard the messages sent by non-trusted processes. This countermeasure is analogous to what UIPI implements to guarantee process privileges. It should be noted that, however, UIPI is not provided with a whitelist of AVs: therefore, it cannot enforce such a filtering in practice and this defense task should be fulfilled by the AV programs.

We elaborate more on this strategy in §7, where we define a requirement that a secure message filtering system should at least have.

5 DISABLING ANTIVIRUS SOFTWARE

In this section, we describe how the simulation attack *Cut-and-Mouse* described in §4 can also be effectively applied in other scenarios, and we also attempt to hypothesize how it can be used in future attacks.

In the course of our analysis, we have found a surprisingly simple utilization of synthesized mouse events technique, which would allow an attacker to deactivate some of the most popular AV programs. We start by explaining the reasons for the presence of deactivation functionalities in AVs. Next, we describe the steps to perform the attack, investigate the weakness in detail, and propose a practical solution to fix it.

5.1 Necessity of the AV Deactivation Function

Signature-based detection has been the primary defense method of AVs, and naturally, this technique is efficient only against known malware as it can be bypassed easily, e.g. by obfuscation/packing and polymorphic malware. To minimize this limitation, nearly all current AVs employ some heuristics to detect malware by monitoring behaviors of processes and looking for anomalies. However, this

functionality comes with a price: occurrences of *false positives*. In the context of malware defense, false positive is the situation where an AV software flags a benign executable as malware, and it usually proceeds with termination of the associated process, hence interrupting the user. For example, when a user installs a new software package, the installer may write to system directories, modify the Windows Registry and configure itself to run when the user logs in. The behavioral decision engine of an AV may be confused by these activities, which indeed might look suspicious as they are largely used by malware. Therefore, an AV may prevent the software from being installed correctly. Consequently, some vendors recommend the users to turn off their AV temporarily for a successful installation of their benign application, for instance [27]. Moreover, some special software may require AV to be disabled while running, for instance [12]. As a result, AV companies provide users with a switch that can be used to deactivate the real-time protection for different periods of time, ranging from a short period, such as 2 minutes, to longer periods, such as 2 hours, or until the computer reboots. Of course, the ability to "freeze" an AV might lure attackers to abuse this functionality to bypass malware detection, hence, AVs should offer ways to ensure that this functionality can be disabled only by authorized users.

5.2 Stopping Real-time Protection

In our second attack, *Ghost Control*, we show how an attacker can disable the AV protection by simulating legitimate user actions to activate the Graphical User Interface (GUI) of the AV program, and then to "click" the turn-off button. The proposed attack comprises two phases. The first phase, *Collect*, is performed off-line by the malware author. In this phase, the developer collects the required pieces of information about the user events to be simulated to successfully disable the AV. This set of information consists of (i) x and y coordinates on the screen; (ii) which mouse button to be simulated; and (iii) length of time to wait until the next menu is available. Please note that the mouse coordinates should lie in the correct area on the screen for this attack to work. In addition, these values would change from victim to victim, or even in the same host, as the screen dimensions vary or would differ under various resolutions. Therefore, the malware author needs to collect the correct locations of the menus of all the major AVs under different display settings to increase the effectiveness. For example, this would require the attacker to install the target AVs in virtual environments with different screen dimensions to collect the necessary data. Once *Collect* phase is completed, malware authors embed the information into the sample to be used during the attack (or, alternatively, they store these pieces of information on a server and deliver them on a request made by the malware).

The second phase of the attack, *Control*, is the actual malicious step which starts immediately after the infection. On the victim machine, *Control* begins the reconnaissance phase to determine the installed AV product(s) and obtain the screen dimensions. Using this information, *Control* prepares the event sequence to be simulated to turn off the AVs, by using the information stored during *Collect* phase, and synthesizes the required user inputs accordingly. Alg. 2 illustrates the part of *Ghost Control* that is responsible for the turning off of the installed AV program.

Algorithm 2 *Ghost Control* Attack: Disable Real-Time Protection of AV with Simulated Events.

```
1: global EventSequenceDatabase
2: function TurnOffProtection
3:     antivirus ← GetInstalledAV()          ▷ AV to deactivate.
4:     events ← GetEventSequenceFor(antivirus)
5:     for all e ∈ events do
6:         Simulate(e)
7:     return Success
```

As a consequence, the range of functionalities that *Ghost Control* enables to malware authors is very large, some having a high impact: for instance, once the real-time scanning is stopped, malware can be instructed to use *Ghost Control* to drop and execute any malicious program from its Command and Control (C&C) server.

5.3 Mitigation

In order to develop a robust defense against this vulnerability, we need to understand the root causes behind this vulnerability. Our analysis shows that there are two reasons why *Ghost Control* is able to deactivate the shields of several AV programs:

(**W**-i) *AV Interface with Medium IL.* Processes related to the AV main interfaces that manage these defense systems run in such a way that they are accessible from processes that run without administrative privileges. It is therefore possible to send "messages" from any process to these process, e.g., mouse click events, without any restriction.

(**W**-ii) *Unrestricted Access to Scan Component.* The scanning components of vulnerable AVs do not require the user to have administrative rights to communicate to them, *e.g.,* they can receive a `TURN_OFF` message from any process. Consequently, *Control* can initiate and control the reaction which involves accessing this critical component of AVs.

(**W**-ii) is actually a more critical vulnerability than (**W**-i). In fact, if an AV software has (**W**-ii), then malware can skip interacting with the GUI of AVs through (**W**-i) to directly communicate with the AV's scanner component and send a `TURN_OFF` message. This is in fact only a practical limitation: for instance, in our experiments (see §6), we have noticed that AV12 employs CAPTCHA mechanisms to verify that the user really wants to turn-off the protection. Even if we assume the CAPTCHA is a solid measure against automated attacks[3], however, malware can still bypass the CAPTCHA verification by directly accessing the scanner component due to (**W**-ii).

To mitigate the root causes of the failure of the affected AVs, we propose the following solution:

(**F**-i) *AV Interface with High IL.* AVs should run the main GUI interface with administrative privileges. By doing so, AVs would not receive the the messages of *Control* or any other malware since UIPI would drop the unauthorized messages.

(**F**-ii) *Restricted Access to Scan Component.* AVs should design and develop their scan components in such a way that accessing it would require the user to have administrative rights.

[3]We note that CAPTCHA can actually be bypassed using other means, e.g. with CAPTCHA solving services, but they might not always be applicable.

In the next section, we discuss and share the results of our experiments, which show that (i) some AVs are vulnerable to *Ghost Control* (ii) the proposed measures are actually employed by some AVs that, therefore, are not vulnerable to the *Ghost Control* attack. From that evidence, we conclude that (i) these attacks are able to circumvent several off-the-shelf AVs (ii) the proposed mitigation is both effective and practical to use in real-world systems.

6 EXPERIMENTAL RESULTS

To demonstrate the impact of the exploitation of the vulnerabilities described in §4 and §5, we developed proof-of-concept prototype of the attacks, and tested them against some commercial AVs. To report our findings, we first describe the test environment.

6.1 Test Environment

We conducted all experiments on a Virtual Machine (VM) running Windows 10 Pro x64 Version 1809 (OS Build 17763.437). After a fresh installation of Windows 10 OS, we updated the system and created a snapshot of a template VM. Next, in each run of the experiment, we restored VM to the snapshot and installed the latest version of the AV software to be tested (available at the time of this writing), which was usually determined by the installer application downloaded from the vendor's website.

The list of the AV programs that we would test in the experiments was determined from the product list published by AV-TEST[4], an independent company which tests AV products of 34 vendors. Most of the software that we decided to test are certified as "top product" in the latest test results of AV solutions for Windows users, which is available at [5]. In addition, we also added some other AV programs to our test set due to their popularity. After the selection procedure, our test set contained 13 AV programs, including most of the AV products of notable vendors.

6.2 Bypassing Protected Folders Feature via Simulated Inputs

In this section, we report the test results where *Cut-and-Mouse* attack is run against AVs. Before we continue, we share the results of some attacks that were detected by AVs.

6.2.1 Attacks Detected by AVs. We first verified whether AVs are able to detect and block known attacks aimed at bypassing the anti-ransomware module. In the first experiment, we injected a malicious DLL into a trusted application, where the DLL would start encrypting the default files protected by AVs. As expected, all of the 13 AVs in our dataset detected this technique, and suspended (or sometimes killed, *e.g.,* `AV13`) the injected trusted application before the first write operation, as DLL injection is one of the oldest attack techniques.

The next experiment was aimed at maliciously controlling a trusted application to save encrypted content to protected files. In this attack, we instructed a ransomware program implemented in C# language to launch the trusted application using `Process.Start` method. As expected, this attack is also not effective as the trusted application is created as a child process of the ransomware, which is not trusted, and therefore blocked by AVs.

[4] AV-TEST, https://www.av-test.org

Lastly, we executed a ransomware with elevated privileges while protected folders feature of AVs were active. The sample, instead of using our *Cut-and-Mouse* technique, is designed to directly encrypt and overwrite the files in `Documents` and `Pictures` folders. Again, all AVs in our dataset detected the attack and blocked the malicious operations, which shows that protected folders feature of AVs is immune to ransomware having admin privileges.

In the next section, we describe the technical requirements for the successful exploitation of *Cut-and-Mouse* attack, and our implementation.

6.2.2 Technical Requirements. Successfully performing *Cut-and-Mouse* attack requires a trusted application that should be available on the victim's machine. Furthermore, this specific trusted application should possess the capabilities to: (i) be started from command line; (ii) accept file paths as argument; (iii) edit/manipulate files; and (iv) receive inputs from clipboard. We have discovered that the best candidate that fulfills all these requirements is the `Notepad` application, since it is one of the most commonly-used built-in Windows application, and it is digitally signed[5], therefore, whitelisted by AV programs. In addition, file size limit of `Notepad` is 56 MB on Windows 7, while it can open documents with size more than 512 MB on Windows 8.1. To send data to from a ransomware sample to `Notepad` application, we exploit Windows Clipboard, which stores objects that can be shared between all running applications. The memory area to store these objects are allocated using `GlobalAlloc` function. On 32-bit systems, virtual memory of a process is limited with 2GB, which also determines the maximum capacity of the clipboard. This gives us a sufficiently large memory space to store encrypted and encoded data, so makes the clipboard suitable to use as a swap area in our attack.

6.2.3 Implementation. We implemented a prototype of *Cut-and-Mouse* in C# language, using .NET Framework version 4.6.1. The prototype synthesizes only keystrokes as input simulation, for which, `SendInput` is employed.

Our prototype implements Alg. 1 and works as follows. First, all of the files in the target directory are enumerated using `Directory.GetFiles`, and the files with the target extensions are filtered. Namely, in the experiments, we targeted the following file extensions: `.docx`, `.xlsx` and `.png`. Next, using `Clipboard.SetText`, ransomware copies the command `attrib.exe -r targetPath\*.*` to the clipboard, where `targetPath` is replaced with the absolute path of the target directory. We instructed the ransomware program to simulate keystrokes `Win+R` to open the `Run` window, and `Ctrl+V` and `ENTER` to run the copied command. This step ensures that the read-only attribute was removed from the target files.

Next, for each file, our *Cut-and-Mouse* prototype proceeds as follows. Firstly, the file is read using `File.ReadAllBytes` and then, using `AesCryptoServiceProvider`, the content of the file is encrypted in memory. After this, the byte stream is converted into printable text using Base64 encoding, and copied to the system clipboard. As previously discussed, our prototype uses `Notepad` as $App_{trusted}$, so it executes `Win+R` command, sleeps 500ms while waiting for the `Run` window to open, and then pastes the command

[5] The digital signature of Notepad, as is the case for many built-in Windows applications, is not embedded in the binary but can be found in the appropriate catalog file.

`notepad.exe targetFile` into the `Run` window, where `targetFile` is replaced with the absolute path of F_{target}. At this step, the prototype sleeps for an additional 500ms to ensure that `Notepad` window is opened – this window displays the contents of the file. Next, the prototype sends the keystrokes `Ctrl+A` to select all the text in the `Notepad` window and `Ctrl+V` to paste the clipboard data into it, which replaces the selected content with the ciphertext. Here, the prototype performs one final sleep of 500ms to ensure that all the data are correctly pasted into `Notepad`. To save the file, `Ctrl+S` command is sent to `Notepad`, which effectively overwrites the file with the encrypted data. Finally, `Alt+F4` command is sent to close `Notepad`.

6.2.4 Test Results of Cut-and-Mouse Attack. After installing the AV software on the VM snapshot, we placed decoy files in the Documents and Pictures folders of the user – these are both protected folders, hence protected from ransomware attacks. Next, we run our *Cut-and-Mouse* prototype and checked the effect of the attack on the files.

The results of our tests are shown in the second column of Table 1. In particular, the results demonstrate the effectiveness of the *Cut-and-Mouse* attack, which was able to bypass seven AV programs and encrypt the files in the protected folders. The other six AVs were not tested against *Cut-and-Mouse* (denoted by n.t.), as they contain a more critical vulnerability which we report in the next section.

To the best of our belief, *Cut-and-Mouse* is a new attack that controls legitimate applications for malicious purposes via simulated user inputs. The evidence that even the latest AV products cannot detect this attack suggests that this new attack type can cause more damages if used by real-world attackers with different –and possibly creative– ideas to perform powerful exploitation of systems.

6.3 Controlling Real-Time Protection of AVs

In order to demonstrate the feasibility of our attack in §5, we implemented the prototype of *Ghost Control* in C# language, using .NET Framework version 4.6.1. To collect the coordinates of the mouse on the screen, the prototype uses `GetCursorPos()` API. For synthesizing keystrokes, mouse motions, and button clicks, `SendInput()` API is used. Between each simulated mouse clicks, the prototype sleeps for 500ms to ensure that the next menu on the GUI is available to be selected.

6.3.1 Collecting Coordinates to Disable AVs. After installing the target AV, we started *Collect* phase and performed cursor movements towards the tray icon area as to select and click the AV icon[6] and used AV's GUI to disable the real-time scanning using the provided menus. During this procedure, we recorded the (x, y) coordinates of the cursor and the types of clicks that we had performed until the protection was disabled, *i.e.,* AV's security notification appeared. For instance, the output of *Collect* while a real user disables `AV8` on a VM with screen resolution set to 4096x2022, is as follows:

[6]For the sake of proof-of-concept, we did not implement a function to detect AV's icon among the tray icons. Actual malware would need to do that, for example, by checking window titles to find AV's icon, but this is not a difficult routine.

Table 1: Evaluation of AV products. Check marks in *Weak Self Protection* column denotes that the AV product was successfully disabled by *Ghost Control*. No further test are performed on AVs that are found to have weak self protection. Check marks in *Weak RW Detection* column denotes that our *Cut-and-Mouse* could bypass the AV product and encrypt the protected files.

Product	Weak Self Protection	Weak RW Detection
AV1		✓
AV2		✓
AV3	✓	n.t.
AV4	✓	n.t.
AV5		✓
AV6		✓
AV7		✓
AV8	✓	n.t.
AV9		✓
AV10	✓	n.t.
AV11	✓	n.t.
AV12	✓	n.t.
AV13		✓

```
Left Click,  x=1868, y=992 // Show Tray Icons
Right Click, x=1866, y=952 // Open AV's Menu
Move Cursor, x=1860, y=873 // Change Settings Submenu
Left Click,  x=1700, y=877 // Real-Time Scan Settings
Left Click,  x=1315, y=430 // Turn-off Button
Left Click,  x=1280, y=555 // Verify Turn-off
```

Figure 3: Output of Collect, which sniffed the real user actions while disabling `AV8`.

For the duration of the deactivation, we used the default values suggested by AVs to freeze their functions. The minimum length is usually set to be 15 minutes, which is a sufficient time frame to successfully conduct an effective attack. Here, the attackers could also select an option that gives them a longer time-period.

6.3.2 Stopping Real-Time Protection. Using the information obtained from *Collect*, we instrumented the recorded actions and parameters into *Control*, which is used to exploit the specific AV that we tested in each experiment. Next, we run *Control* and waited until all the events are simulated.

If *Control* attack succeeds, a warning window appears which notifies the user that the computer is not protected. In some experiments, we even went further and simulated mouse clicks to remove this notification window, which would be expected from a real-world malware. This shows how this class of attacks can be further extended to perform potentially more powerful malicious actions.

As shown in the first column of Table 1, during our experiments on 13 AV products, we detected that 6 AVs could be efficiently deactivated by *Ghost Control* using our attack in §5. According to a recent report by OPSWAT [24], the market share of AVs that are

vulnerable to *Ghost Control* is at least 23%[7]. Furthermore, 4 of these AVs have been frequently rated as "top product" in the reports of AV-TEST. It is surprising for us that such a critical vulnerability, arguably one of the worst that an AV might have, is found in such a large share of AVs.

In the experiments in which *Ghost Control* was not able to successfully disable the AV, we noticed that this was due to User Account Control (UAC) prompt, which uses MIC. In these cases, after *Ghost Control* generated a click event to turn-off protection, UAC notification appeared, which always runs with High IL. However, since *Ghost Control* is a Medium IL process, it was not be able to bypass UAC verification successfully.

7 DISCUSSION

Secure composability is a well known problem in security engineering. It challenges developers to ensure that security properties enjoyed by individual software components are preserved when the components are put together. It also challenges them to demonstrate that the components together give stronger security assurances than just the mere sum of their original properties. This rarely happens in practice, and the opposite is quite often true. Components that, when taken in isolation, offer a certain known attack surface do generate a wider surface when integrated into a system. Intuitively this seems obvious. Components interact one another and with other parts of the system create a dynamic with which an attacker can interact too and in ways that were not foreseen by the designer. An attacker can, for example, uses a component as an oracle or replay its output to impersonate it while interacting with another.

This is exactly what we have found happening to mechanisms like UIPI and AV software. They provide a robust defense when tested individually against a certain target, but the attacks that we demonstrate in this paper show that their combination reveals new vulnerabilities. We draw two considerations from it.

First, in complex systems it is essential to control the message-flow between security critical components. This is actually enabled by Microsoft via UIPI. It allows messages flowing from sender applications to receiver applications only when the integrity level of the first is not less than the integrity level of the second. In principle, UIPI enables a good defence mechanism, but the problem is that integrity levels do not reflect trust: they merely indicate when an application runs with administrative right (high), in standard mode (medium), or in a sandbox (low). The authority who decides which level an application takes is generally the operating system, and sometimes the user, after a request from the application. It may be, like in the scenario that we illustrated in Section 5, that developers do not implement that request.

This is against what Microsoft Driver Security Guidance suggests [20]: "*It is important to understand that if lower privilege callers are allowed to access the kernel, code risk is increased. [..] Following the general least privilege security principle, configure only the minimum level of access that is required for your driver to function.*". We think that the process which controls the status of the anti-malware and AV's kernel module should be designed to require 'high' IL. Our findings show that several anti-malware companies either failed to follow this guidance or have misjudged the minimum level requested for their security, or did not diversify enough between kernel and non-kernel modules.

Secondly, and this is linked to our finding in Section 4, relying only on integrity levels is not sufficient to ensure system security. This does not surprise, since UIPI has been designed to protect processes, and in fact anti-malware applications top-up their defence strategy relying on whether an application is whitelisted, that is, *trusted*. Only trusted applications can *e.g.,* access protected files. But, our findings have revealed a dissonance here: medium integrity level applications, like `Notepad`, are considered trusted and thus allowed to *e.g.,* access protected files. But an application with medium integrity level, that is running with standard user rights, does not necessarily behave in a benign manner. As we showed for the case of `Notepad`, medium but untrusted applications, such as malware, can have their actions looking like be trusted by using the application as a puppet; in so doing, they can bypass the anti-malware guard.

We think that a better defence is to combine the integrity levels and the trust label used by anti-malware. We state it as the following principle:

Security Principle 1. *Messages between applications should be allowed only when the sender has at least the same integrity level as the receiver* and *and the sender is at least as trusted as the receiver.*

Principle 1 reminds the renowned Bell and La Padula Model on messages-flow between different security "clearence" levels [6] (see also [28]). But it is not exactly the same, since we cope with "security" instead of confidentiality. Attempting a formalization of Principle 1, components should be classified by "security levels", made of two elements: the UIPI "integrity levels", (I = [admin , user, or sandbox], ordered) and the anti-virus software's "trust levels" (T = [digitally signed / whitelisted, not digitally signed / not whitelisted], also ordered). Principle 1 suggests a policy saying that an application of security level (I, T) should not accept messages coming from applications of security level (I', T') when $(I' < I)$, or when $(I' \geq I)$ but $(T' < T')$.

In conclusion, we believe that applying Principle 1 would have prevented receiving `SendInput` from effecting whitelisted applications that has a potential to be exploited, *e.g.,* `Notepad`. One should, however, evaluate whether this may also broke some of the existing automation software solutions. A conclusive statement about this would require to perform a wide spread test on automation applications. It also had fostered AV vendors take measures not only to protect the system, but also to protect their AV programs against other supposedly trusted applications, in addition to conventional malware attacks against AV products. A practical fix is to configure AV kernel module to require admin rights to be accessed. In this regard, it might be helpful to monitor `SendInput` API and block all simulated keyboard and mouse events dispatched to AV program although the problem of understanding whether a low-level event, such as an interrupt, has been generated by a human or not might be difficult to solve in general.

[7] We were not able to calculate the exact statistics as the shares of the 3 AVs that we could stop are consolidated into "Others".

8 RELATED WORK

In this section, first we review existing attacks involving simulated inputs to perform malicious actions. Next, we outline previous research on the security of antivirus software.

8.1 Attacks Related to Input Simulation

Input simulation is the practice of programmatically synthesizing input events, such as mouse clicks or key strokes, which are typically performed by the user. This section describes some the most powerful existing attack techniques that make use of input simulation.

8.1.1 Ghost Clicks. In [30], Springall *et al.* developed a proof-of-concept malware to manipulate votes in Estonian Internet Voting system. On infected clients, the malware simulates keyboard inputs to activate the electronic identifier (e-ID) of voters and submit a vote in a hidden session that is invisible to the voters.

Recently, under a different threat model, in [18] Maruyama *et al.* demonstrate a method to generate tap events on touch screens of smart phones using electromagnetic waves. In this scenario, the victim's device can be forced to pair with a malicious Bluetooth device once it gets in the range of the attackers. Even if the victim denies the pairing by choosing `CANCEL` in the security prompt, the attacker can alter this selection and make the OS to recognize user input as `CONNECT`.

Pay-per-click advertising systems are also vulnerable to fake clicks, which is known as Click Fraud [31]. In these systems, the advertisers get paid according to the number of clicks on advertisements. By generating fraudulent clicks on the ads, a malicious advertiser can increase its payment.

Perhaps the attack closest to the one described in this paper is *Synthetic Clicks* [21], credited to Patric Wardle [11]. Exploiting a bug in macOS OS, the attacker could send programmatically-created mouse clicks events to security prompts that would result in vertical privilege escalation. This way the attacker could cause any damage, including retrieving all of the user's passwords stored in the keychain. Our attacks, *Cut-and-Mouse* and *Ghost Control*, target Windows OS, do not rely upon a bug in the OS, and can be used to instruct a privileged application to perform different malicious operations.

8.1.2 Reprogramming USB Firmware. In [22], Nohl *et al.* demonstrated that it is feasible to modify the firmware of a USB device, for instance a USB stick, to behave like a keyboard. Known as BadUSB, this technique works by reprogramming the device's firmware in order to type commands on the victim's computer. When plugged into a computer, the malicious USB device can simulate the key strokes of the user, for example, type and execute a script which downloads and runs a malware.

8.1.3 Shatter Attack. In [3], Paget describes a weakness in Windows OS that allow a process to inject arbitrary code into another process and execute. The "shatter attack", a term coined by Paget, works as follows: first, the malware copies the code-to-be-injected to the clipboard. Next, it sends `WM_PASTE` message to target process to paste the clipboard contents into a text field on the GUI of the target process. At this point, the malicious code has been moved onto the memory space of the target process. To execute this code, the malware process sends another window message, a carefully crafted `WM_TIMER` message, which causes a jump to the address of the malicious code. The main difference with our attacks is the presence of the malicious code during the injection, while with *Cut-and-Mouse* we use and control a privileged application as a "puppet" to perform various operations without injecting new code into the target process memory.

8.2 Previous Research on Security of AVs

Traditionally, AVs have been in the target of security researchers due to their incomparable importance. Since AV vendors mostly utilize blacklisting as the main defense technique, many researchers investigated this area. For instance, [9] and [29] analyzed the feasibility of evade detection via obfuscation.

Another significant research topic about AVs is the implementation related vulnerabilities. To name a few examples: [15, 16, 25, 26, 32]. That said, the discoveries in this field mostly involve the bugs in the AV software, rather than a flaw in their design or threat model.

Finally, in [2], Al-Saleh and Crandall developed a technique to determine if the target AV is up-to-date using side channel analysis, allowing the attacker to learn which signatures exists in virus database of the victim.

9 CONCLUSIONS

Antivirus programs (AVs) have become one of the *de facto* computer security standards. Recently, they have also integrated ransomware detection modules. There has been quite an attention to this class of malware, given its world-wide impact; therefore, interested to know how current AVs can mitigate the threat, we started to dive into the matter. What we found is indeed surprising. Despite the great attention to security that AV companies put into their products, the security issues we discovered are in the interaction between OS defences and AV defences.

Precisely, we found that it is possible for a malicious program to (i) turn off AV's real-time scanning protection feature; and (ii) bypass anti-ransomware protected folder solutions by misusing whitelisted applications to encrypt user data. To this end, we have discussed and provided two proof-of-concept programs, *Ghost Control* and *Cut-and-Mouse*, which were able to either disable several off-the-shelf AVs or bypass their anti-ransomware feature. We believe that the two issues can be fixed and avoided in the future, but this requires software developers to have a general understanding of what caused them. We stated that understanding in our Principle 1.

One could question whether such attacks can be detected by the human user's seeing, *e.g.,* the mouse icon clicking here and there. However, we believe that making security dependent on the user's reaction is fundamentally a wrong design choice, as user may indeed enlarge the attack surface; in addition, malware can perform these attacks when the user is not using the computer, *e.g.,* through some heuristic based on user's activities. Thus a better mitigation solution would be aimed at understanding whether keyboard and mouse events come from a legitimate user or whether instead they are synthesized by a (malicious) program. In a sense, discerning such situation is what malware is already trying to achieve, namely

understanding if it is running in a sandbox, *e.g.*, using reverse Turing tests to detect the presence (or absence) of a human, – this further reinforces the analogy of attackers and defenders are each learning from others. However, before that discernment becomes possible, OS and AV defences have to cooperate better. At the root of our findings there is a misalignment between two different concepts: that of integrity levels used by the OS, and that of trusted applications on which instead AV defences rely upon. They have not been conceived to work together and, at a higher level, they have to be harmonized. This is indeed what our Principle 1 means to achieve. We will attempt a synthesis of the two concepts by developing a proof-of-concept component that implements it, thus creating a test-bed for the validity of the Principle itself, which is one of our future research works.

ACKNOWLEDGMENTS

This work was partially funded by European Union's Horizon 2020 research and innovation programme under grant agreement No 779391 (FutureTPM) and by Luxembourg National Research Fund (FNR) under the project PoC18/13234766-NoCry PoC.

REFERENCES

[1] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. 2019. A taxonomy of software integrity protection techniques. In *Advances in Computers*. Vol. 112. Elsevier, Cambridge, MA, USA, 413–486.

[2] Mohammed I. Al-Saleh and Jedidiah R. Crandall. 2011. Application-level Reconnaissance: Timing Channel Attacks Against Antivirus Software. In *Proceedings of the 4th USENIX Conference on Large-scale Exploits and Emergent Threats (LEET'11)*. USENIX Association, Berkeley, CA, USA, 9.

[3] Chris Paget (alias Foon). 2002. Exploiting design flaws in the Win32 API for privilege escalation. Retrieved May 15, 2019 from https://web.archive.org/web/20060904080018/http://security.tombom.co.uk/shatter.html

[4] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. arXiv:cs.CR/1801.08917

[5] AV-TEST. 2019. The best antivirus software for Windows Home User. Retrieved June 10, 2019 from https://www.av-test.org/en/antivirus/home-windows/

[6] D. E. Bell and L. J. La Padula. 1976. *Secure computer system: Unified exposition and Multics interpretation.* Technical Report ESD-TR-75-306. Mitre Corporation.

[7] Battista Biggio and Fabio Roli. 2018. Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2154–2156. https://doi.org/10.1145/3243734.3264418

[8] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based Malware Detection System for Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2046614.2046619

[9] Mihai Christodorescu and Somesh Jha. 2004. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 34–44.

[10] Ian Goodfellow, Patrick McDaniel, and Nicolas Papernot. 2018. Making Machine Learning Robust Against Adversarial Inputs. *Commun. ACM* 61, 7 (June 2018), 56–66. https://doi.org/10.1145/3134599

[11] Andy Greenberg. 2019. Another Mac Bug Lets Hackers Invisibly Click Security Prompts. Retrieved June 10, 2019 from https://www.wired.com/story/apple-macos-bug-synthetic-clicks/

[12] IT Services of Mitchell Hamline School of Law. 2017. Technology Notice – Disable Antivirus before using Examplify. Retrieved May 31, 2019 from https://mitchellhamline.edu/technology/2017/12/03/technology-notice-disable-antivirus-before-using-examplify/

[13] S. Josefsson. 2006. *The Base16, Base32, and Base64 Data Encodings.* RFC 4648. RFC Editor. http://www.rfc-editor.org/rfc/rfc4648.txt http://www.rfc-editor.org/rfc/rfc4648.txt.

[14] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 769–780.

[15] Joxean Koret. 2014. Breaking Antivirus Software. Retrieved June 10, 2019 from http://joxeankoret.com/download/breaking_av_software_44con.pdf

[16] Joxean Koret. 2016. AV: Additional Vulnerabilities. Retrieved June 10, 2019 from https://www.hoystreaming.com/wp-content/uploads/2016/03/hb_bilbo.pdf

[17] Joxean Koret and Elias Bachaalany. 2015. *The Antivirus Hacker's Handbook.* John Wiley & Sons, Indianapolis, IN, USA.

[18] S. Maruyama, S. Wakabayashi, and T. Mori. 2019. Tap 'n Ghost: A Compilation of Novel Attack Techniques against Smartphone Touchscreens. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 628–645.

[19] Alana Maurushat. 2013. *Disclosure of Security Vulnerabilities: Legal and Ethical Issues.* Springer-Verlag London, London.

[20] Microsoft. 2019. Driver security checklist. Retrieved June 10, 2019 from https://docs.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist

[21] NIST. 2017. NVD – CVE-2017-7150. Retrieved June 10, 2019 from https://nvd.nist.gov/vuln/detail/CVE-2017-7150

[22] Karsten Nohl, Sascha Krißler, and Jakob Lell. 2014. BadUSB—On accessories that turn evil. Retrieved May 15, 2019 from https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf

[23] Working Group Dual Use of the Flemish Interuniversity Council. 2017. Guidelines for researchers on dual use and misuse of research.

[24] OPSWAT. 2019. Windows Anti-malware Market Share Report. Retrieved June 10, 2019 from https://metadefender.opswat.com/reports/anti-malware-market-share#!/

[25] Tavis Ormandy. 2015. Analysis and Exploitation of an ESET Vulnerability. Retrieved June 10, 2019 from https://googleprojectzero.blogspot.com/2015/06/analysis-and-exploitation-of-eset.html

[26] Tavis Ormandy. 2016. How to Compromise the Enterprise Endpoint. Retrieved June 10, 2019 from https://googleprojectzero.blogspot.com/2016/06/how-to-compromise-enterprise-endpoint.html

[27] TaxSlayer Pro. 2017. Quick Start Manual. Retrieved June 10, 2019 from http://downloads.taxslayer.com/online/2017-Quick-Start-Manual.pdf

[28] John Rushby. 1986. *The Bell and La Padula Security Model.* Computer Science Laboratory, SRI International, Menlo Park, CA. Draft Technical Note.

[29] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation.

[30] Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. 2014. Security Analysis of the Estonian Internet Voting System. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 703–715.

[31] Kenneth C. Wilbur and Yi Zhu. 2009. Click Fraud. *Marketing Science* 28, 2 (2009), 293–308.

[32] Feng Xue. 2008. Attacking Antivirus. Retrieved June 10, 2019 from https://blackhat.com/presentations/bh-europe-08/Feng-Xue/Presentation/bh-eu-08-xue.pdf

[33] Feng Xue. 2008. Attacking The Antivirus.

[34] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

[35] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA '10)*. IEEE, Piscataway, New Jersey, US, 4.