# Cut-and-Mouse and Ghost Control: Exploiting Antivirus Software with Synthesized Inputs

ZIYA ALPER GENÇ and GABRIELE LENZINI, University of Luxembourg
DANIELE SGANDURRA, Royal Holloway, University of London

To protect their digital assets from malware attacks, most users and companies rely on antivirus (AV) software. AVs' protection is a full-time task against malware: This is similar to a *game* where malware, e.g., through obfuscation and polymorphism, denial of service attacks, and malformed packets and parameters, tries to circumvent AV defences or make them crash. However, AVs react by complementing signature-based detection with anomaly or behavioral analysis, and by using OS protection, standard code, and binary protection techniques. Further, malware counter-acts, for instance, by using adversarial inputs to avoid detection, and so on. In this cat-and-mouse game, a winning strategy is trying to anticipate the move of the adversary by looking into one's own weaknesses, seeing how the adversary can penetrate them, and building up appropriate defences or attacks. In this article, we play the role of malware developers and anticipate two novel moves for the malware side to demonstrate the weakness in the AVs and to improve the defences in AVs' 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 whitelisted 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 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*. We tested these two attacks on 29 AVs, and the results show that 14 AVs are vulnerable to *Ghost Control* attack while all 29 AV programs tested are found vulnerable to *Cut-and-Mouse*. Furthermore, we also show some weaknesses in additional protection mechanisms of AVs, such as sandboxing and CAPTCHA verification. We have engaged with the affected AV companies, and we reported the disclosure communication with them and their responses.

CCS Concepts: • **Security and privacy → Malware and its mitigation**;

Additional Key Words and Phrases: Antivirus, ransomware, evasion, vulnerability, synthesize inputs, simulate mouse

**ACM Reference format:**
Ziya Alper Genç, Gabriele Lenzini, and Daniele Sgandurra. 2021. Cut-and-Mouse and Ghost Control: Exploiting Antivirus Software with Synthesized Inputs. *Digit. Threat.: Res. Pract.* 2, 1, Article 4 (February 2021), 23 pages.
https://doi.org/10.1145/3431286

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 No. PoC18/13234766-NoCry PoC.
Authors' addresses: Z. A. Genç and G. Lenzini, University of Luxembourg; emails: {ziya.genc, gabriele.lenzini}@uni.lu; D. Sgandurra, Royal Holloway, University of London; email: daniele.sgandurra@rhul.ac.uk.

2576-5337/2021/02-ART4
https://doi.org/10.1145/3431286

## 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 antivirus (AV) on their devices. In fact, AVs are believed to be one of the best protection solutions, specifically against malware. They 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 databases [2], but over the years malware authors have spent great effort in trying to evade AVs' detection, e.g., through obfuscation and polymorphism [40] or evasion [3], or by disabling or crashing the AV [18, 37]. AVs and malware are engaged in a *cat-and-mouse* game: attacks based on polymorphism are typically mitigated by some form of anomaly or behavioral detection [8, 39, 39] while evasion attacks are 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 AVs' behavioral detection using, for instance, adversarial inputs [7], and AVs will incorporate robust mechanisms to mitigate the effects of these inputs [11, 15]. In this *cat-and-mouse* game, one party tries to anticipate the move of the other. We believe AVs should be the party more involved in this thinking-ahead, for instance by questioning whether certain principles, or certain best practices on which their defences rely upon, are valid, and under which assumptions and limitations they are so.

Taking this stand, let us consider the best practice of using a whitelist instead of a blacklist. The advantage of whitelists has been largely discussed elsewhere (e.g., see Reference [31]), but what we question here is whether AVs, and OSs, make their security dependent too much on them by assuming that whitelisted built-in applications of OSs, like `Notepad` and `Paint`, can never do any harm. To test this hypothesis, in this work, we create a malware that can control those whitelisted applications like puppets by instructing them to perform malicious operations. Then, we verify whether, dressed this way, our malware can bypass all the defences that AVs and OSs put in place to protect files, e.g., have them in the so called *Protected Folders*. It turns out that we can, and we named this attack *Cut-and-Mouse*.

In our second hypothesis, we question whether AVs go one step further, and assume that users have the choice to decide whether to set the offered protections off and on, without considering whether it is really the user doing so, or someone (or something) just simulating the user's behaviour. Thus, we tested whether off-the-shelf AVs can be disabled (i.e., can be turned off or frozen) by a malware that mimics user inputs through synthesized keyboard and mouse events. We expected that AVs would have enforced some forms of integrity and authentication checks on inter-process communications, and user access control to verify the legitimacy of the received inputs. It turns out that a great deal of them do not. Our attack, named *Ghost Control*, managed to disable several AVs by spoofing requests to their main graphical interface.

**Problem Statement:** This article 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) *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 themselves are not protected from being misused by other applications. This trust is therefore

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

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 on their behalf, thus to encrypt user data. Similarly, personal files of users might be irrevocably destroyed by a wipeware.

In this article, we play one move more of the game. We suggest a new principle that, if followed and properly implemented, render AVs' and OSs' resilient to the attacks (*Cut-and-Mouse* and *Ghost Control*) that we have found.

*Novel contribution w.r.t. previous work.* This article builds up on previous research of ours, a conference article published in Reference [10]. There, we introduce for the first time *Cut-and-Mouse* and *Ghost Control* attacks and tested 13 AVs against them. In this work, we extend that research with several novel insights: (1) we performed further experiments on 16 new AVs and we present more experimental results, so that this article's conclusions about AVs' resilience or vulnerabilities against *Cut-and-Mouse* and *Ghost Control* attacks are based on all available consumer level AV products, 29 AVs in total; (2) we provide an improved version of *Cut-and-Mouse* attack, which is more efficient and destructive; (3) we comment on our process of responsible disclosure where we informed AVs of our findings, and we report the answers of the AVs that have answered us; (4) then, inspired, or provoked, by some early replies that dismissed our attacks as non-critical (some other however took it seriously and fixed the vulnerabilities or promised to), we first show how our *Cut-and-Mouse* and *Ghost Control* can overcome even some advanced defences that AVs assume to be secure, such as running apps in a sandbox and using CAPTCHA as a method of distinguishing humans from malware; in so doing, we reveal the limits of other two security best practices; (5) we also discuss a new defense mechanism against our *Ghost Control* malware that acts at OS-level, and that is therefore capable of preventing our malware from disabling an AV's real-time protection even if the AV does not take the precautions we suggested to avoid the attack in the first place; (6) finally, we compare our *Cut-and-Mouse* and *Ghost-control* attacks with known attacks, namely, the Shatter Attack [29] and the Synthetic Clicks [23], in anticipation of future comparative discussions about the nature of event-based attacks and of the vulnerabilities that they exploit.

*Outline.* In Section 2, we report the disclosure process and responses of the vendors. We give preliminary information about AVs and process protection in Section 3. The threat model that we assumed in this work is stated in Section 4. In Section 5, our first attack, *Cut-and-Mouse*, which encrypts files in protected folders via synthesized user inputs, is described. Next, we explain our second attack, *Ghost Control*, to control the real-time protection of AVs in Section 6. We also provide a detailed report of our dataset, test environment and results of experiments in Section 7. Furthermore, extended analysis of two auxiliary security measures used by AVs are given as a case study in Section 8. The root cause of the issues discovered in this article, and a theoretical result we obtain are discussed in Section 9. Previous attacks in literature, and their comparison to the attacks we discovered is presented in Section 10. Finally, in Section 11, we conclude the article.

## 2 ETHICAL CONSIDERATIONS: COORDINATED AND RESPONSIBLE DISCLOSURE

This research has ethical concerns of dual use research: our findings can be used to improve the security model of AVs as well as to attack them. Aware of the risk, we adhere to an ethical code of conduct [25]: 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 article. We also follow a practice of responsible disclosure [20]: We have dutifully engaged with the affected AV companies to inform them about our findings, ensuring they knew about our research. We have shared with them whatever they needed to replicate our attacks and to gain insights to fix the vulnerabilities that we think are the root cause of the success of the attacks. All the affected AV companies have been dutifully informed. We told them that, we believe, is their the responsibility to fix whatever needs to be fixed, but we leave to them to judge the severity and the impact of our research on their products and clients.

Table 1. The Process and Results of the Responsible Disclosure with the Affected AV Vendors

| Vendor | Dedicated Channel | Encrypted Email | Disclosure Platform | Disclosure Date | Response Time | Current Decision |
|---|---|---|---|---|---|---|
| V4 | ✓ | ✓ | ✗ | 30.10.2019 | 1 day | Released a fix |
| V5 | ✗ | ✗ | ✗ | 09.09.2020 | No Response | |
| V6 | ✓ | ✗ | ✗ | 09.09.2020 | <1 day | Working on |
| V7 | ✓ | ✗ | ✗ | 30.10.2019 | 1 day | Rebutted |
| V8 | ✓ | ✗ | ✗ | 09.09.2020 | 3 days | Won't Fix |
| V12 | ✗ | ✗ | ✗ | 09.09.2020 | <1 day | |
| V14 | ✗ | ✗ | ✗ | 09.09.2020 | No Response | |
| V16 | ✓ | ✗ | ✓ | 30.10.2019 | 1 day | Not Prioritized |
| V20 | ✓ | ✓ | ✗ | 09.09.2020 | No Response | |
| V24 | ✓ | ✓ | ✓ | 10.09.2020 | 4 days | |
| V26 | ✗ | ✗ | ✗ | 09.09.2020 | No Response | |
| V27 | ✓ | ✓ | ✗ | 30.10.2019 | 5 days | Fixed |
| V28 | ✗ | ✗ | ✗ | 30.10.2019 | No Response | |
| V29 | ✗ | ✗ | ✗ | 30.10.2019 | 1 day | Working On |

V$x$ denotes the vendor of AV$x$.

We have not forced nor nudged them to react by using our research as a leverage. Such a mission is outside to our ethical stand-point. In any case, we will not reveal the names of AVs as with some of them.

That said, we think useful to report some data about how many AVs we have found vulnerable, and how they have responded to our attempts, to contact them. Table 1 summarises the situation for the companies we identified that AV products of 14 vendors are vulnerable to our attacks. Ten of 14 vendors have engaged in a conversation with us or have answered somehow.

Please note that we have only engaged with the vendors of the 14 AVs we discovered vulnerable. Therefore, we do not know whether these attacks are successful (or not) on other AVs, but we do hope that the results of our research motivate other AV vendors to perform a similar security analysis and, in case, adopt countermeasures.

*Communication Method.* We strove to communicate over official channels to inform the vendors, and share the technical details of the issues, proof-of-concept materials, as well as potential mitigation techniques. As we can see from Table 1, V16 and V24 use independent platforms for vulnerability disclosure. We filled application forms and marked the vulnerability type as a *critical*. Four vendors—V4, V20, V24, and V27—accept encrypted emails and their PGP keys are available on the company websites. We used this communication method and sent encrypted emails to these addresses (except V24, which was reached using an independent platform). V6 and V8 accepts unencrypted emails (V6 publishes a PGP key, but it does not match the email address dedicated for vulnerability disclosure). V7 provides a dedicated web page for vulnerability reports, so we filled the form on the page with a description of the issue. V5, V12, and V26 do not provide dedicated channels to report vulnerability, therefore, we engaged with these companies using their support mail. Last, V14, V28, and V29 do not have any suitable channel to reach out, thus we filled the contact forms at the company websites as a last resort.

*Reactions.* Six of 14 vendors replied to our disclosure in 1 day (hereafter day means calendar day). However, no vendor could give us an exact time for a fixed release. V4 is the only vendor that informed us that it released the fixed version. V27 worked with us to fix the vulnerability and implemented the patch, however, did not inform us if it has released the fixed version. Quite surprisingly, vendor of AV16 acknowledged that V16 is vulnerable but they considered its severity as low, and did not put a patch for it their priority list. V8 also informed us that the issue will not be addressed by V8 as it should be fixed in the OS level. V6 and V29 replied that they acknowledged the vulnerability and are working on the issue, however, did not inform us about any release plan. V7 rebutted our initial disclosure; we replied back and wait for their response. We have not heard back from V12 and V24 about the issue. We attempted to contact V28 several times. In our last attempt, V28 closed the ticket that we described the vulnerability and never replied back to us. We have not received any reply from V5, V14, V20, and V26. V4 an V27 offered to publish our names on the company websites. Furthermore, V4 offered us a bounty for our vulnerability disclosure. Microsoft has been informed about our *Cut-and-Mouse* attack since the beginning. They acknowledged the weakness in Windows 10 and informed us that an investigation is going on to find the root cause of the weakness and to formulate an efficient fix.[2]

## 3 BACKGROUND

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

### 3.1 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 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 article, we use the term *trusted applications* when referring to the applications that has write access to protected folders, either granted by AV vendor, by the OS or added by the user.

### 3.2 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. Mandatory Integrity Control (MIC) defines four values for Integrity Level (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. User Interface Privilege Isolation (UIPI) also utilizes ILs and blocks window messages flowing from a process with lower IL. For example, calls to `SendMessage`

[2]Note that, in this specific case, it is not possible to anonymize the name of the affected vendor as the root of the vulnerability that enables one of our attacks lies in the Operating System running the AVs (Windows) rather than the AVs. At the time of the writing, the responsible disclosure period, 90 days, has elapsed so we feel compelled to share the details publicly.

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 Section 10.

## 4 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 article 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 *second-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 folders, while the second one (*Ghost Control*) is aimed at disabling AVs' real-time protection.

## 5 CUT-AND-MOUSE: ENCRYPTING PROTECTED FOLDERS

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

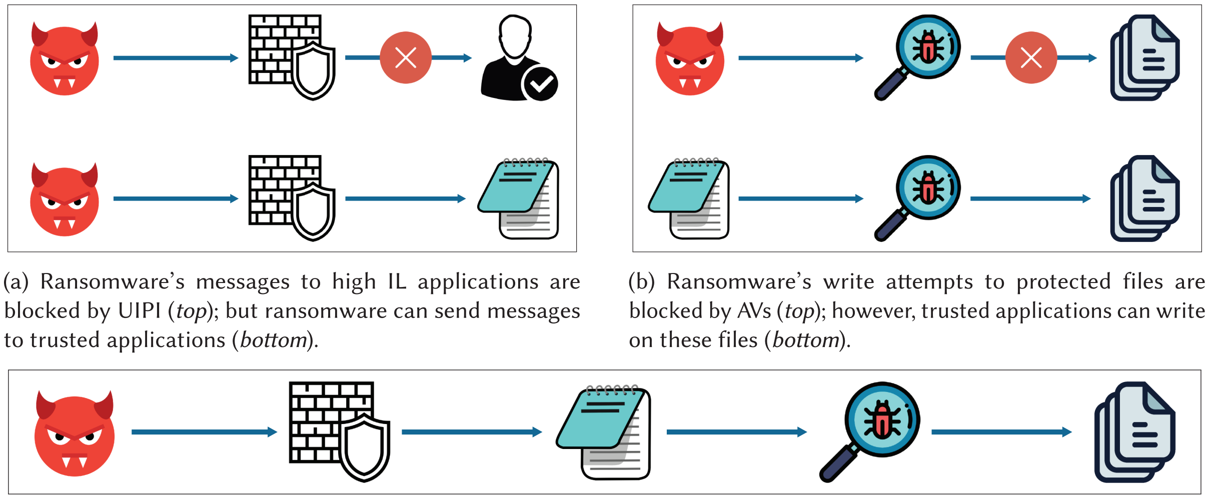
### 5.1 Disharmony between UIPI and AVs

As explained in Section 3, 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 that would be seen in a typical malware attack. For instance, as we detail in Section 7, 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 Section 3, protects processes that run with administrative privileges from malware.

Nonetheless, we have discovered two entry points for an attack that enable 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 Figure 1(a), 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 Some Process Messages*: AVs do not monitor synthesized clicks or key press events flowing into the trusted applications: as depicted in Figure 1(b), 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 Figure 1(c) 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, trusted applications can write on these files (*bottom*).

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

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

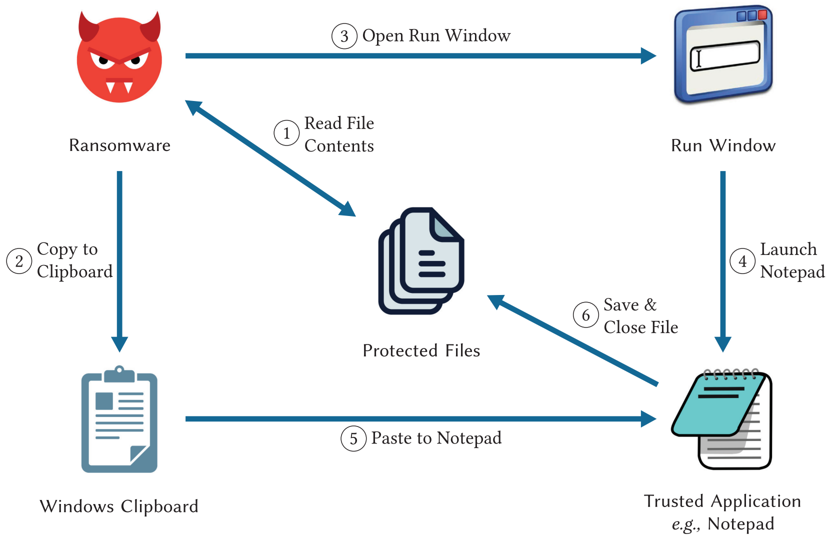


Fig. 2. Bypassing anti-ransomware protection of AVs by using inputs programmatically synthesized by ransomware to control a trusted application.

## 5.2 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 Figure 2. First, 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.[3] 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 [14], 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 that are synthesized programmatically by the ransomware to control $App_{trusted}$.

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

```
 1: function Cut-and-Mouse($App_{trusted}$)                    ▷ Application to Control.
 2:     $FileList$ ← EnumerateTargetFiles()
 3:     for all $f \in FileList$ 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 $Success$
```

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 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 that are required for the *Cut-and-Mouse* attack to be successful. First, the step *Open Run Window* (3) in Figure 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, then 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. Second, as noted in Footnote 3, the step *Read File Contents* depicted in (1) in Figure 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

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

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.

### 5.3 Proposed 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 messages flowing to a trusted process and block (or discard) the messages sent by non-trusted processes. This countermeasure is analogous to what UIPI already implements to guarantee process privileges. It should be noted that, however, UIPI is not provided with whitelists 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 Section 9, where we define a requirement that a secure message filtering system should at least have.

## 6 GHOST CONTROL: DISABLING ANTIVIRUS SOFTWARE

In this section, we describe how the simulation attack *Cut-and-Mouse* described in Section 5 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 nearly half of the consumer AV programs, including some popular products. We start by explaining the reasons for the presence of deactivation functionalities in AVs. Next, we describe the steps to perform the second attack proposed in this article (*Ghost Control*), investigate the weakness in detail, and propose a practical solution to fix it.

### 6.1 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 [30]. Moreover, some special software may require AV to be disabled while running, for instance [13]. 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 min, to longer periods, such as 2 h, 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.

### 6.2 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 is performed off-line by the malware developer. 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 data collection is completed, malware author embeds that information into the malware executable to be used during the attack (alternatively, malware can download the required information from a remote server at the time of attack).

The second phase of the attack is the actual malicious step, which starts immediately after the infection. On the victim machine, malware performs a reconnaissance work to determine the installed AV product(s) and obtain the screen dimensions. Next, malware prepares the event sequence to be simulated to turn off the AVs, and synthesizes the keyboard and mouse events accordingly. Algorithm 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 as EvSecDB
2: function TurnOffProtection
3:     antivirus ← GetInstalledAV()                              ▷ AV to deactivate.
4:     events ← EvSecDB.GetEventSequenceFor(antivirus)
5:     for all $e \in events$ 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.

## 6.3 Proposed Mitigations

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, *Ghost 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 Section 7), 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,[4] however, malware can still bypass the CAPTCHA verification by directly accessing the scanner component due to (**W**-ii).

It is worth noting the subtle difference between {(**E**-i), (**E**-ii)} and {(**W**-i), (**W**-ii)}. We believe (**E**-i) and (**E**-ii) are due to an optimistic, or defective, or misdirected threat analysis: it appears as those threats have not been considered, leaving the system undefended against them. However, (**W**-i) and (**W**-ii) resemble the results of committing a vulnerable code to the repository of AV software, and could be avoided by a security assessment in the development process.

To mitigate the root causes of the failure of the affected AVs, we propose a solution based on the following principles:

- (**F**-i) *Elevated AV Interface.* AVs should run the main GUI interface with administrative privileges. By doing so, AV processes will have high IL, and AVs would not receive the messages of *Ghost 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.

*Mitigation at OS Level.* Windows platform provides a tool to monitor critical components and applications, including virus protection, firewall, and OS updates. This tool, called Windows Security Center (WSC), reports the security status of the system to Action Center. For example, WSC detects if an AV program is installed, and continuously checks if the AV is turned on and up-to-date. If, for some reason, real-time protection of the AV stops, then WSC informs Action Center, which notifies the user and provides an interface to take a remediation action.

We propose to adapt this architecture to detect and prevent (**W**-i) and (**W**-ii) as follows. AV programs can already be integrated to Action Center by registering themselves with WSC. During registration, the path of the main executable of the AV program is supplied to WSC along with the product name and other pieces of information. WSC can use these data to perform security checks on the AV executable, in particular, WSC can

(1) prevent the registration of the AV if the AV's interface is configured to run with medium IL;
(2) auto-escalate the integrity of the AV process to high; or
(3) set the security descriptors of an AV components to default value so that accessing them requires system or administrator privileges.

The first option, preventing installation of the AV, can be viewed as undesirable, especially considering the availability of the second and third options. However, it should be noted that a mitigation that includes raising an error would ultimately allow the developers to be aware of the vulnerability, and might lead them to discovering other issues that would remain hidden otherwise.

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 these attacks are able to circumvent several off-the-shelf AVs; and the proposed mitigation is both effective and practical to use in real-world systems.

## 7 EXPERIMENTAL RESULTS

To demonstrate the impact of the exploitation of the vulnerabilities described in Sections 5 and 6, we developed three proof-of-concept prototypes for the attacks, and tested them against consumer products of 29 AV companies. In this section, we detail the dataset and test environment of our experiments, and report our findings.

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

## 7.1 Dataset and Test Environment

The list of the AV programs that we would test in our experiments was determined from the reports of independent organizations that test AV products. We populated our initial list with the AVs from the recently published reports of AV-TEST [5] and AV-Comparatives [4]. The initial list had AVs from 35 vendors, however, some vendors discontinued their consumer AV product, or were not available for download. In the end, our dataset contained 29 AV programs from world wide vendors.

We conducted all experiments on a Virtual Machine (VM) running Windows 10 Pro x64 Version 1903 (OS Build 18362.30) OS. After a fresh installation of Windows 10, we updated the system and created a snapshot of a template VM. Next, in each run of the experiment, we restored the 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. Finally, we updated the database of the AV software to obtain the latest signature definitions and heuristics.

## 7.2 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 Dynamic-Link Library (DLL) into a trusted application, where the DLL would start encrypting the default files protected by AVs. As expected, all of the 29 AVs in our dataset detected this technique, and suspended (or sometimes killed, e.g., `AV17`) 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.

Last, 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. Therefore, using existing attack techniques, all attempts to write into protected folders are blocked by the AVs.

## 7.3 Encrypting Files in Protected Folders via Simulated Inputs

In this section, we report the test results where attack is run against AVs. First, we describe the technical requirements for the successful exploitation of attack, and our implementation.

*7.3.1 Technical Requirements.* Successfully performing 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 that are larger than 512 MB on Windows 8.1. File size limit of `Notepad` comes with Windows 10 is not documented by Microsoft, but the trend suggests that it should be higher than the limits of previous versions. To send data 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

[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.

with 2 GB, 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.

*7.3.2 Implementation.* We implemented a prototype of 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 Algorithm 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 prototype proceeds as follows. First, the file is read as binary 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 500 ms while waiting for the `Run` window to open, and then pastes the command `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 500 ms 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 500 ms 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`.

*7.3.3 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.

At the end of each run, the decoy files were overwritten with the pasted data successfully. The results demonstrate the effectiveness of the *Cut-and-Mouse* attack, which was able to bypass *all 29 AV programs* in our test set and encrypt the files in the protected folders. 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.

## 7.4 Destructive Cut-and-Mouse: Wiping Files in Protected Folders

Although *Cut-and-Mouse* attack is effective on AVs, the limitations of using `Notepad` forms a performance barrier when the file size noticeably increases. To remove this bottleneck, we will use another built-in Windows application, `Paint`, as intermediary.

`Paint` also satisfies all the technical requirements in Section 7.3.1 with a couple of exceptions. First, only some image files are accepted as a file argument. `Paint` raises an error when the user tries to open a `.PDF` document, for example. Second, `Paint` only accepts a valid image from clipboard. If the image in clipboard is corrupted, then it cannot be pasted to `Paint`.

The first limitation can be overridden easily by adding a file extension explicitly, which would allow `Paint` to write to any files. The second limitation, however, makes it difficult to build an "operational" (i.e., fully working)

```
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
```

Fig. 3. Console output of the application that sniffed the real user actions while disabling AV27.

ransomware, as it requires the implementation of a reversible encoding technique to transform arbitrary data to a valid image format. Instead, for the scope of this research, we demonstrate another type of malware, known as *wipeware*, able to overwrite user's files with a randomly generated image to destroy them permanently.

As in Section 7.3.2, *Cut-and-Mouse* wipeware also starts with collecting target files and removing their read-only attributes. Next, the wipeware prototype creates a random bitmap image, which has the same size of the largest file. The random image is copied to clipboard by calling `Clipboard.SetImage`. Then, for each target file, the wipeware performs the following tasks in order: (i) synthesize Win+R, programmatically type `mspaint.exe` in the Run window, and synthesize Enter to open `Paint`; (ii) synthesize Ctrl+V to paste the random image from clipboard after `Paint` windows appears; (iii) synthesize Ctrl+S to save the file, which would open up the `File Save` dialog; (iv) programmatically type the full path of the file and synthesize Enter; (v) synthesize Enter to confirm the overwrite message box; (vi) synthesize Alt+F4 to close `Paint`. By sleeping 500 ms between each step to ensure all operations are carried out correctly, our *Cut-and-Mouse* wipeware could destroy each decoy file in a few seconds.

## 7.5 Controlling Real-time Protection of AVs

To demonstrate the feasibility of our attack in Section 6, 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()` Application Programming Interface (API). For synthesizing keystrokes, mouse motions, and button clicks, `SendInput()` API is used. Between each simulated mouse clicks, the prototype sleeps for 500 ms to ensure that the next menu on the GUI is available to be selected.

*7.5.1 Collecting Coordinates to Disable AVs.* After installing the target AV, we performed cursor movements towards the tray icon area as to select and click the AV icon[6] and used AV's Graphical User Interface (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, Figure 3 shows the console output of the application we used to collect mouse coordinates while a real user disables AV27 on a VM with screen resolution set to $1920 \times 1080$.

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 min, 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.

*7.5.2 Stopping Real-time Protection.* Using the collected coordinates of the AV's menus and buttons, we instrumented the recorded actions and parameters into our *Ghost Control* prototype, which is used to exploit the specific AV that we tested in each experiment. Next, we run the *Ghost Control* prototype and waited until all the events are simulated.

[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 2. Evaluation of AV Products

| Product | IL of GUI | Utilizes UAC | Vulnerable to *Ghost Control* | Vulnerable to *Cut-and-Mouse* |
|---|---|---|---|---|
| AV1* | Medium | ✓ | | ✓ |
| AV2* | Medium | ✓ | | ✓ |
| AV3 | Medium | ✓ | | ✓ |
| AV4* | Medium | | ✓ | ✓ |
| AV5 | Medium | | ✓ | ✓ |
| AV6 | Medium | | ✓ | ✓ |
| AV7* | Medium | | ✓ | ✓ |
| AV8 | Medium | | ✓ | ✓ |
| AV9* | Medium | ✓ | | ✓ |
| AV10* | Medium | ✓ | | ✓ |
| AV11 | Medium Plus | | | ✓ |
| AV12 | Medium | | ✓ | ✓ |
| AV13* | Medium | ✓ | | ✓ |
| AV14 | Medium | | ✓ | ✓ |
| AV15 | Medium | ✓ | | ✓ |
| AV16* | Medium | | ✓ | ✓ |
| AV17* | Medium | ✓ | | ✓ |
| AV18 | High | | | ✓ |
| AV19* | Medium | ✓ | | ✓ |
| AV20 | Medium | | ✓ | ✓ |
| AV21 | Medium | ✓ | | ✓ |
| AV22 | High | | | ✓ |
| AV23 | High | | | ✓ |
| AV24 | Medium | | ✓ | ✓ |
| AV25 | High | | | ✓ |
| AV26 | Medium | | ✓ | ✓ |
| AV27* | Medium | | ✓ | ✓ |
| AV28* | Medium | | ✓ | ✓ |
| AV29* | Medium | | ✓ | ✓ |
| Tested: 29 | | | Vulnerable: 14 | Vulnerable: 29 |

Check marks under *Vulnerable to Ghost Control* denotes the AV products that were successfully disabled by *Ghost Control*. Next, *Vulnerable to Cut-and-Mouse* column reports the AV programs that could not detect the encryption of protected files by *Cut-and-Mouse*. AV label with an asterisk (*) indicates that that AV was evaluated in our previous work.

If *Ghost Control* attack succeeds, then a warning window appears that 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 Table 2, during our experiments on 29 AV products, we detected that 14 AVs could be efficiently deactivated by *Ghost Control* using our attack in Section 6. According to a recent report by OPSWAT [26], the market share of AVs that are vulnerable to *Ghost Control* is more than 29%.[7] Furthermore, six of these AVs have been frequently rated as "TOP PRODUCT" in the reports of AV-TEST, and three of them received three stars

[7] We were not able to calculate the exact statistics as the shares of the 10 AVs that we could stop are consolidated into "Other."

(best rating) from AV-Comparatives. 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 two factors. First and foremost, User Account Control (UAC) prompt, which uses MIC stopped *Ghost Control* attack. 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. Second, as shown in Table 2, five AVs always run with medium plus IL or high IL. Consequently, UIPI filters and drops the events that our *Ghost Control* prototype synthesizes and sends to these AVs.

## 8 SECURITY ANALYSIS OF AUXILIARY MEASURES

During the experiments, we were confronted with two additional security measures, namely, sandboxing and CAPTCHA verification, which protected AVs from *Ghost Control* in cases where the GUI of the tested AV was vulnerable. In this section, we will look at these measures and explain how we were able to bypass them.

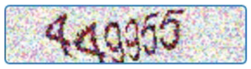
### 8.1 Insecure Sandboxing Methods

In the security context, sandboxing is a mechanism to run an unknown application in a controlled environment, isolated from the host. The main purpose of employing sandbox in AVs is to prevent previously unseen malware from damaging the system, which would evade the signature-based detection otherwise. Although the high level understanding of sandboxing is common to all AVs, the implementation details might vary between different vendors. In addition, AVs do not publicly share the inner workings of their sandboxes, so we can only guess the capabilities of sandboxes from their whitepapers and AV settings.

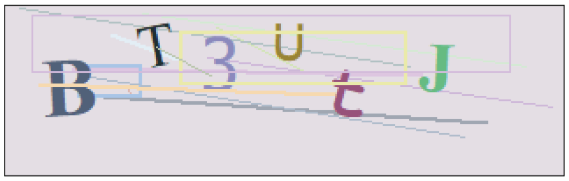
Most vendors in the AV industry supply sandbox products for their business-level customers, usually as a gateway device to be integrated into the network. Some AVs provide cloud-based sandboxes for home users where unknown files are submitted for analysis. For example, AV6 offers its users a cloud service to analyze files with certain extensions. With that said, we could identify that only AV1, AV2, and AV7 let users run programs in a virtual, isolated environment on their computers. Other AVs might also have a built-in sandbox technology, but according to our observations, they do not expose any settings, show any notifications indicating a sandbox, nor advertise any information among the products' features.

According to our experiments, both AV1 and AV2 execute each unknown program in a sandbox at the first run. This automatic execution seems to be time limited, and takes around 30 s. After that, the programs are automatically started in the host environment without the sandbox restrictions. Users can also run programs in the sandbox without a time limit using the context menu. From program outputs and error messages, we infer that both AVs create a virtual file system where the programs being tested cannot access the files on host, even for reading. The programs can access to the Internet though, and the trusted files they download can be saved outside the sandbox. Furthermore, programmatically synthesized events, such as simulated mouse clicks and key strokes cannot reach outside the sandbox. By sleeping for 30 s in total and meanwhile performing a benign task like printing to console, our *Ghost Control* prototype was found harmless by the sandbox of AV1 and AV2, however, it was stopped by UIPI.

Differently, AV7 does not apply a time limit for the automatic sandboxing at the first run. Moreover, it allows the programs being tested to read the actual files on the host. From the company website and program interface, we infer that the sandbox of AV7 prevents any running application from writing to any file or registry by placing function hooks, sending inter-process messages and window messages, or synthesizing keyboard events. The enforcement is applied even to the processes that run with admin privileges, therefore, an unknown process' device driver installation is also denied by the sandbox.



(a) AV10 generates CAPTCHA codes that contain only numeric characters.



(b) AV29 generates CAPTCHA codes that consist of lowercase and uppercase letters, and numbers.

Fig. 4. CAPTCHA codes generated by AV10 (left) and AV29 (right). The images in the figure are in 1:1 scale.

However, we noticed that the sandbox of AV7 is developed in such a way that mouse clicks are not filtered, and therefore a malware can easily escape the sandbox. In other words, it could synthesize mouse clicks using `SendInput`. During our experiments, even if all unknown applications were automatically run in the sandbox, mouse events synthesized by our prototype were not filtered by the sandbox. Combined with the vulnerable GUI of AV7, our commands received by AV7 and we could stop the real-time protection. Our conclusion from this observation is to add a fix to the weak sandbox by including mouse events to the filtered operations, in addition to fixing the vulnerable GUI of AV7.

## 8.2 Passing Human Verification

Completely Automated Public Turing Test to Tell Computers and Humans Apart (CAPTCHA) is a challenge-response test to determine if the user is a human or not [35]. CAPTCHA is a widely adopted technology on the Internet to distinguish humans from computers. In our experiments with 29 AVs, we identified that two AV programs, AV10 and AV29, utilized CAPTCHA images in their program flow, as illustrated in Figure 4.

AV10 shows a CAPTCHA image and asks the code therein when Shutdown Protection menu item is clicked (which terminates the main AV process and stops protection) to ensure that the GUI interaction is engaged by a human, not a malware. The vendor of AV10 indicates that this measure is employed to prevent automated shutdown by malware. However, no verification is performed when Disable Protection menu item is clicked—in which case, the AV program continues to run, but protection service is stopped. As a result, *Ghost Control* could easily disable AV10.

When testing our attack to disable AV29, during the final step of the turn-off sequence, this AV generates a CAPTCHA image and displays on the screen to verify that the request comes from a genuine user. The user must enter the code correctly for AV protection to be turned off. In our first try, our *Ghost Control* prototype failed to turn off AV29 as it could not enter the CAPTCHA code. To overcome this limitation, we enhanced the prototype with the ability to partially capture the screen that contains the CAPTCHA code. Next, the prototype sends the captured image to an external user, who can solve the CAPTCHA and sends the correct CAPTCHA code back to our prototype, which synthesizes the code to AV29, and completes the turn off sequence.

Although our method for solving CAPTCHA codes might look like a naïve and impractical solution that does not scale, please note that cybercriminals have at least two alternatives, as follows:

- Capture the CAPTCHA image and dispatch to Command and Control (C&C) server where a CAPTCHA-solver program is running, and return the code to the malware. The latest advancements in Machine Learning techniques allow to develop highly accurate text-CAPTCHA solvers [38]. The CAPTCHA images shown in Figure 4 might be solved by an automated software.
- Use CAPTCHA-solver services available to solve CAPTCHAs for affordable prices with high success rates to make the attack scalable and profitable [22].

## 9 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 article 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 6, that developers do not implement that request.

This is against what Microsoft Driver Security Guidance suggests [21]: "*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 that 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.

Second, and this is linked to our finding in Section 5, 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 Reference [32]). 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 to 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. However, some vendors, in their responses to our vulnerability disclosure, noted that *"[...] With the introduction of Patch Guard it is very difficult to protect win32k.sys (which provided access to UI functions)"*. They also noted that *"[...] There exist several techniques to bypass User Account Control (UAC) to gain admin privileges; this is not a fault of the AVs and exacerbates the problem."* We agree with this remark, although in this article, we have considered the UAC as part of the Trusted Computing Base (TCB), as otherwise other attacks would also be possible.

## 10 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.

### 10.1 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.

*10.1.1 Ghost Clicks.* In Reference [34], 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 Reference [19] 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 [36]. 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 article is *Synthetic Clicks* [23], credited to Patric Wardle [12]. 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 AVs, not OS, do not rely upon a bug in the OS, and can be used to instruct a trusted application to perform different malicious operations.

*10.1.2 Reprogramming USB Firmware.* In Reference [24], 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

Table 3. Comparison of *Cut-and-Mouse* and *Ghost Control* to Relevant Attacks that Synthesize Window Messages

| Characteristics | Ghost Control | Cut-and-Mouse | Synthetic Clicks [23] | Shatter Attack [29] |
|---|---|---|---|---|
| Exploits a Bug in OS | No | No | Yes | No |
| Modifies Target Process | No | No | No | Yes |
| Utilizes Clipboard | No | Yes | No | Yes |
| Requires a Text Edit Field | No | Yes | No | Yes |

technique works by reprogramming the device's firmware 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 that downloads and runs a malware.

*10.1.3 Shatter Attack.* In Reference [29], 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.

## 10.2 Comparison to Previous Attacks

*Cut-and-Mouse* and *Ghost Control* are two novel attacks on AVs, of which the main principle is to simulate user commands by programmatically synthesizing mouse and keyboard events. As we reviewed above, there are other techniques in the literature that shares similar behaviour. Table 3 illustrates the characteristics of these attacks and compares to that of *Cut-and-Mouse* and *Ghost Control*.

First, *Ghost Control* attack does not require a bug to exist in the OS, instead, it targets the applications that do not use the privileges provided by the OS, similar to Shatter Attack. In contrast, Synthetic Clicks [23] depends on a bug in the OS. Second, differently from Shatter Attack, which performs arbitrary code execution, *Ghost Control* only uses the functions exposed within the GUI of the target application. Therefore, *Ghost Control* leaves no trace in memory, while Shatter Attack modifies the target process and leaves artifacts that can be detected in the memory dumps. In a sense, *Ghost Control* attack can be considered as puppeteering the target application while Shatter attack is more close to poisoning the target. That said, *Ghost Control* needs exact coordinates of the screen to successfully work, while Shatter Attack is independent of the target system's display.

Similar to Shatter Attack, *Cut-and-Mouse* utilizes system clipboard and needs the target application to have a text field. However, similar to *Ghost Control*, the impact of *Cut-and-Mouse* attack is also limited to the functionality of the target applications, i.e., `Notepad` and `Paint`. Consequently, *Cut-and-Mouse* technique is naturally suited to damage files and, therefore, can be exploited to perform ransomware, wipeware or similar destructive attacks.

Finally, *Cut-and-Mouse* and *Ghost Control* attacks do not require the creation of a child process or remote thread, unlike the attacks detected by the AVs. Those attacks involve some actions, such as injecting a DLL payload or process launching, that can be effectively monitored and analyzed by the AVs. On the contrary, we believe that it would require more efforts to identify if a key stroke or a mouse click event—which are the only two building blocks of our *Cut-and-Mouse* and *Ghost Control* attacks—are part of a malicious action.

### 10.3 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, References [9] and [33] analyzed the feasibility of evade detection via obfuscation. Another significant research topic about AVs is the implementation related vulnerabilities. To name a few examples, References [16, 17, 27, 28, 37]. 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 Reference [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.

## 11 CONCLUSIONS

AV programs have become one of the *de facto* computer security standards. Several companies trust AVs to protect their assets without questioning how AVs do their job. In their turn, AVs have their assumptions, we presume, and trust their security mechanisms be solid. Sometimes, we learned, also that trust is not questioned further. This is probably necessary in the fast-paced cat-and-mouse game in which AVs and malware are engaged, always running one after the other; but we, as researchers, can question the robustness of certain assumptions. In particular, we questioned whether built-in whitelisted applications can undetectably be manipulated and instructed to do harm to user files. For instance, we tried to see whether they can be used to encrypt a file's content or to wipe it out. We also questioned whether AV's real-time scanning protection feature can be turned-off by a malware that simulates mouse and keyboard events without being caught while doing so. Surprised ourselves, we succeeded in proving that these vulnerabilities exist for quite a number of AVs. What we found is indeed surprising in general, considering that almost all AVs have today ransomware detection modules.

The security issues we discovered reveal vulnerabilities both in the extension in which certain security mechanisms are supposed to operate, and in the they way in which the interaction between the OS and the AV defences is believed to work. The vulnerabilities we discuss in this article are therefore not implementation flaws. To give substance to our findings, we have designed and implemented two proof-of-concept programs, *Ghost Control* and *Cut-and-Mouse*, which are able to fully disable the real-time protection of several consumer grade AVs, and/or to bypass their defences in protection of user files against threats like ransomware. We tested them against the current most comprehensive list of consumer level AVs products. Not all of them are vulnerable, but for those that are, we also speculated about possible fixes. These require software developers to have a general understanding of what caused them. We stated that understanding in a potentially new, or at least renewed, *security principle*.

One could question whether such *Cut-and-Mouse* and *Ghost Control* attacks can, after all, be detected by the human user who sees, e.g., the mouse pointer moving and clicking here and there. Perhaps, users can be puzzled. Still they are likely not be able to react promptly to stop the attack: users are notoriously bad on implementing security measures, because security is not their primary goal. Thus making security dependent on the user's reaction to something strange on her screen is fundamentally not a solution and it does not give more security guarantees.

We have also found that *Cut-and-Mouse*'s and *Ghost Control*'s working principle, that of using mouse and keyboard events, works even when AVs run them in a sandbox, revealing that the sheer use of a sandbox is not sufficient to protect a system from certain malware: mouse clicks are not filtered and therefore can escape the sandbox.

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.

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 Principle 1 itself, which is one of our future research works.

## 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, 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, 9.

[3] 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. Retrieved from https://arxiv:cs.CR/1801.08917.

[4] AV-Comparatives. 2020. Malware Protection Test March 2020. Retrieved from https://www.av-comparatives.org/tests/malware-protection-test-march-2020/.

[5] AV-TEST. 2020. The best antivirus software for Windows Home User. Retrieved from https://www.av-test.org/en/antivirus/home-windows/windows-10/february-2020/.

[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 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. ACM, New York, NY, 2154–2156.

[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, 15–26.

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

[10] Ziya Alper Genç, Gabriele Lenzini, and Daniele Sgandurra. 2019. A game of "Cut and Mouse": Bypassing antivirus by simulating user inputs. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. Association for Computing Machinery, New York, NY, 456–465.

[11] Ian Goodfellow, Patrick McDaniel, and Nicolas Papernot. 2018. Making machine learning robust against adversarial inputs. *Commun. ACM* 61, 7 (June 2018), 56–66.

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

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

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

[15] 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, 769–780.

[16] Joxean Koret. 2014. Breaking Antivirus Software. Retrieved from http://joxeankoret.com/download/breaking_av_software_44con.pdf.

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

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

[19] S. Maruyama, S. Wakabayashi, and T. Mori. 2019. Tap 'n ghost: A compilation of novel attack techniques against smartphone touchscreens. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19)*. IEEE Computer Society, Los Alamitos, CA, 628–645.

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

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

[22] Marti Motoyama, Kirill Levchenko, Chris Kanich, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. 2010. Re: CAPTCHAs: Understanding CAPTCHA-solving services in an economic context. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, 28.

[23] NIST. 2017. NVD–CVE-2017-7150. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2017-7150.

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

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

[26] OPSWAT. 2020. Windows Anti-Malware Market Share Report. Retrieved from https://www.opswat.com/blog/windows-anti-malware-market-share-report-april-2020.

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

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

[29] Chris Paget. 2002. Exploiting design flaws in the Win32 API for privilege escalation. Retrieved from https://web.archive.org/web/20060904080018http://security.tombom.co.uk/shatter.html.

[30] TaxSlayer Pro. 2017. Quick Start Manual. Retrieved from http://downloads.taxslayer.com/online/2017-Quick-Start-Manual.pdf.

[31] Marcus Ranum and Bruce Schneier. 2011. Schneier-Ranum Face-Off on whitelisting and blacklisting. Retrieved from https://searchsecurity.techtarget.com/magazineContent/Schneier-Ranum-Face-Off-on-whitelisting-and-blacklisting.

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

[33] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*. The Internet Society. https://www.ndss-symposium.org/ndss2008/impeding-malware-analysis-using-conditional-code-obfuscation/.

[34] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, New York, NY, 703–715.

[35] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. 2003. CAPTCHA: Using hard AI problems for security. In *Proceedings of the Conference on Advances in Cryptology (EUROCRYPT'03)*. Springer, Berlin, 294–311.

[36] Kenneth C. Wilbur and Yi Zhu. 2009. Click fraud. *Market. Sci.* 28, 2 (2009), 293–308.

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

[38] Guixin Ye, Zhanyong Tang, Dingyi Fang, Zhanxing Zhu, Yansong Feng, Pengfei Xu, Xiaojiang Chen, Jungong Han, and Zheng Wang. 2020. Using generative adversarial networks to break and protect text captchas. *ACM Trans. Priv. Secur.* 23, 2, Article 7 (Apr. 2020), 29 pages.

[39] 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, 116–127.

[40] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *Proceedings of the International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA'10)*. IEEE, Piscataway, New Jersey, 4.

Received May 2020; revised September 2020; accepted October 2020