Keylogger Detection Using a Decoy Keyboard

Seth Simms, Margot Maxwell, Sara Johnson, and Julian Rrushi(✉)

Department of Computer Science, Western Washington University,
Bellingham, WA 98226, USA
{simmss2,maxwelm,johns782}@students.wwu.edu
julian.rrushi@wwu.edu

Abstract. Commercial anti-malware systems currently rely on signatures or patterns learned from samples of known malware, and are unable to detect zero-day malware, rendering computers unprotected. In this paper we present a novel kernel-level technique of detecting keyloggers. Our approach operates through the use of a decoy keyboard. It uses a low-level driver to emulate and expose keystrokes modeled after actual users. We developed a statistical model of the typing profiles of real users, which regulates the times of delivery of emulated keystrokes. A kernel filter driver enables the decoy keyboard to shadow the physical keyboard, such as one single keyboard appears on the device tree at all times. That keyboard is the physical keyboard when the actual user types on it, and the decoy keyboard during time windows of user inactivity. Malware are detected in a second order fashion when data leaked by the decoy keyboard are used to access resources on the compromised machine. We tested our approach against live malware samples that we obtained from public repositories, and report the findings in the paper. The decoy keyboard is able to detect 0-day malware, and can co-exist with a real keyboard on a computer in production without causing any disruptions to the user's work.

Keywords: Decoy I/O · 0-knowledge anti-malware · Kernel drivers

1 Introduction

Malware development, dispersion, and infection are an ever-present threat to system security and user privacy. Among the most insidious and difficult to detect are 0-day malware, as well as those using code and data structure mutation. With nearly limitless access to system services, drivers, and modules, these types of malware have some distinct advantages over current detection methods – not only must they have been previously encountered and analyzed in order to be caught, but they can interfere with and elude the software trying to track them. Keyloggers are a common component of malware, able to spy on a user's activity and gather information like passwords, account numbers, and credit card numbers.

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
G. Livraga and S. Zhu (Eds.): DBSec 2017, LNCS 10359, pp. 433–452, 2017.
DOI: 10.1007/978-3-319-61176-1_24

There are several methods used by keyloggers to capture information from the keyboard in the Windows operating system. User level keyloggers can implement a software hook that can intercept keystroke events, perform cyclical querying of the keyboard device to determine state changes, or inject special code into running processes that has access to the messages being passed. Kernel level keyloggers work by using filter drivers that can view the I/O traffic bound for a keyboard. There are varying methods of implementing keyloggers, and while many are documented, it is likely that there are some that are not. New keystroke interception techniques are discovered continuously.

Contribution. We present a novel defensive deception approach, which uses a decoy keyboard to receive contact from keyloggers in a way that leads to their detection unequivocally. The decoy keyboard emulates the presence and operation of its real counterpart with the help of drivers in the kernel. The decoy keyboard can be installed on a full production system in active use, and is able to not interfere with the activities of a normal user. The decoy keyboard appears as a standard USB keyboard in the Windows device tree. It is effectively invisible to the user, and indistinguishable from a real keyboard to malware. The defender's objective is to proactively misdirect malware into intercepting keystrokes emulated by the decoy keyboard. An attack surface is created by sending emulated keystrokes to a decoy process (dprocess), which runs in user space. Malware are detected when they use the data that they had intercepted from the decoy keyboard on the compromised computer. Some malware are detected upon contact with the decoy keyboard. The decoy keyboard can be combined with a decoy mouse and a decoy monitor for consistency reasons, using techniques similar to those discussed in this paper.

Novelty. The decoy keyboard is a 0-knowledge detector, meaning that it can detect malware without any prior knowledge of their code and data. 0-day malware, polymorphic malware, metamorphic malware, data-structure mutating malware, are all detectable by this work. A shadowing mechanism enables the decoy keyboard to coexist with its real counterpart on a computer in production. A timing model helps the decoy keyboard expose emulated keystrokes on the attack surface in a realistic and consistent fashion. Data intercepted from a decoy keyboard have the same timing characteristics as data intercepted from a real keyboard. The decoy keyboard is a usable security tool. It runs automatically with very little computational overhead, and requires no user input or any other involvement. Overall, our approach creates an active redirection capability that is effective in trapping malware.

Threat Model. Our approach currently works against keyloggers that run in user space. Kernel-level malware are planned as future work. Malware could use any exploits to land on a computer, and could have any form of internal design to intercept keystrokes. Our approach works independently of the exploitation techniques and the inner workings of malware. We did this work with reference to the Windows operating system, as we were seeking a proof of concept to show

the potential of our approach. Similar principles can be applied to other modern time-shared operating systems as well.

Paper organization. The remainder of this paper is organized as follows. Section 2 describes the proposed approach in four parts, namely modeling human keystroke dynamics, low-level deceptive driver, keyboard shadowing, and a discussion of how our approach attains malware detection. In Sect. 3 we discuss the evaluation of the proposed approach against live malware. In Sect. 4 we discuss related work in malware detection, deception tactics, and keystroke dynamics. Section 5 summarizes our findings and concludes the paper.

2 Approach

2.1 Modeling Human Keystroke Dynamics

Just as we humans have unique patterns in our handwriting (to the level that it can be used for identification), we have similar patterns in the way that we type on computer keyboards. This has been extensively studied [35] as a biometric means of user authentication and identification, with generally good results. However, it is important to note that we are not using this model for authentication, but rather for *generation* of keystrokes. This presents a different set of unique challenges, and eliminates some of the specific considerations needed for user verification. Notably, we are proceeding with the assumption that the malware is not specifically targeting a user or users that they already have acquired a large amount of recorded keystroke data and built complex models for. We used the large body of previous research to help define a model that we could use to send emulated keystrokes that would appear to come from a unique human user, while simultaneously meeting our goals.

One primary goal was having low overhead, since we are working at the driver and operating system level, and additional processing time could add unwanted and unintentional delays that could cause our deception to become visible. Low overhead implies a measure of simplicity in the model, with the additional consideration that a more complex and specific model is difficult to implement without error and likely easier to detect, as well as being unsuitable for the initial stage of research. Of course, model accuracy was also important, keeping in mind that the accuracy of keystroke dynamic authentication may not map directly to our generation of keystrokes. Based on these considerations and requirements, we chose to implement a model consisting of the time between each key in a *digraph*, or two consecutive keys. For example, in the word 'the', there are two digraphs: 'th' and 'he'.

This is similar to the models used by Gaines [33] and Umphress [32] (among others), and has shown to be quite effective in user authentication for text such as passwords. Importantly, Gaines found there was little difference in the digraph times between english text, random words, or random phrases, and the data was found to be normally distributed. A histogram of a single example digraph is shown in Fig. 1.

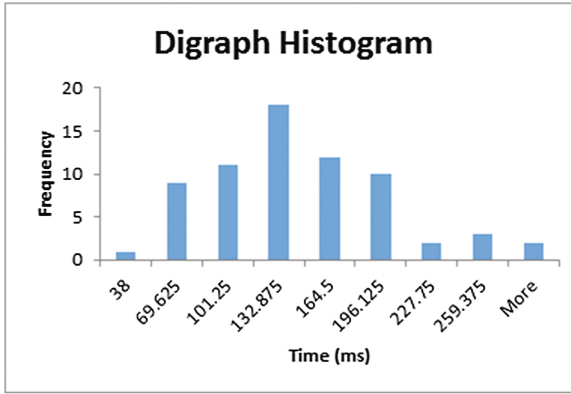

Fig. 1. Distribution of timing data for a single digraph.

The model itself consists of the mean time in milliseconds and the sample standard deviation for each possible key combination. Each unique digraph can be represented as a normal curve with those two values. With a relatively small finite number of possible digraphs, this can be quickly implemented with a lookup table containing the two variables for each digraph, and a statistical function used to generate a random number (for milliseconds of key delay) that falls within the normal curve. Thus we are very likely to get a value that is close to the mean, but occasionally we will get a delay that is less than or more than usual, as is common with human behavior. We developed the model using data from some publicly available datasets [36–38], but for the operation of the software we are using data recorded from individual members of the research group, in order to present a unique user.

While our software is still a prototype, future production and distribution across numerous machines could enable the possibility of malware analyzing data from different sources that are all operating the software, and thereby the ability to expose the deception if they all had nearly identical timing values. Thus we must provide a unique user, and we evaluate and test the software using models created from real people. Working with these datasets exposed us to the issue of outliers when using recorded data. After implementing and testing our model, we discovered that there were several instances of unnaturally long delays between generated keys, due to unrealistically high standard deviation values. This was due to the fact that any sort of pause in the middle of typing, even to the extent of leaving the computer for several minutes, was reflected in the timing data; these outliers were affecting the model and were not representative of what we were trying to model.

We explored two methods of outlier removal, the simple but naive method (removing values that fall outside of a certain number of standard deviations from the mean), and a more robust method using the median absolute deviation (MAD). Leys et al. [39] describes this method in detail, but the results are evident and can be seen in Table 1.

Table 1. Comparison of outlier removal methods for a single digraph

	Baseline	Naive	MAD
Mean	261.7162	158.2817	131.8971
Median	127	124	120.5
Std. dev.	562.0665	142.0196	57.17455

To generate the models used in testing the deceptive driver against real malware, we recorded our own typing data for a full page of text and generated a model for each user, associating every digraph with a mean time in milliseconds and standard deviation. This is also how the decoy keyboard learns the typing profile of a user on a machine in operation. Significant outliers were removed, and digraphs for which there was insufficient data (less than three occurrences) were discarded. When encountering digraphs for which there is no entry in the model, the deceptive driver will use a predetermined mean value with high variance to represent the unpredictability of such rare occurrences; regardless, a user's typing patterns can be identified using a limited number of common digraphs [33], and rare occurrences are not of much use in generating or validating a model. To illustrate the effectiveness of the model in distinguishing users, the timing values for the ten most common digraphs of three of the authors are shown in Fig. 2. Each data point consists of the mean timing value for one of the ten digraphs, where the distance from the center represents the time in milliseconds. We can see that the two users in the middle have roughly the same typing speed, but they have unique characteristics for each digraph. The user on the outside is significantly different just by nature of a slower typing speed.

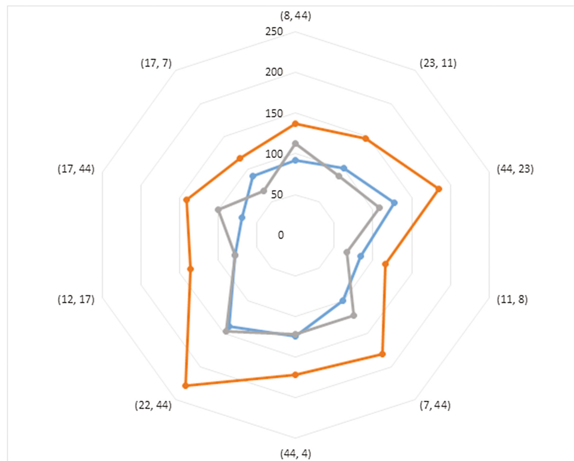

Fig. 2. Comparing the digraph model of three users.

2.2 Low-Level Deceptive Driver

This is the driver that emulates keystrokes. We refer to this driver as the decoy keyboard (DK) driver. It is a customized Windows USB Human Interface Device (HID) function driver, based on an open-source project called VMulti [43], i.e., a virtual multiple HID driver using the Kernel Mode Driver Framework (KMDF). We modified and extended VMulti to turn it into a decoy keyboard driver. KMDF is the same framework used by many vendor drivers for real hardware. It allows us to follow the same requirements and standards as a real keyboard. Thus, it appears in the device tree and Device Manager just as any other device. The location of the decoy function driver within the Windows device tree can be seen in Fig. 3. Normally, a driver does not initiate messages or data traffic itself, but responds to signals either from the hardware device (like keys being pressed) or from the operating system (turning on lights for Caps Lock, etc.).

In our case, with no physical hardware to generate signals, the keystrokes must originate from within the driver itself. That is, keystrokes are processed as normal key events through the driver so as to be indistinguishable from a real keyboard. These events have no explicit time stamps. They are retrieved using a first in, first out queue. However, any software that intercepts or records the events can easily associate a time value to each event for the purposes of analysis. Keystrokes sent without regard to timing would simply go at machine speed, that is, far faster than any human could possibly type. This is why we use the statistical model that we described previously, to regulate the time each event is sent, emulating human typing behavior. The model of a user's typing profile is represented internally as an array that functions as a lookup table, providing the core statistical values for each pair of keys.

On loading, the DK driver processes the statistical model and thus initializes a normal distribution function for each digraph, using the mean and standard deviation in combination with a cryptographically secure random number generator. Keystrokes are sent by the driver one at a time. Each key event is an individual action. Text/keycodes are taken as input and then are processed by the DK driver. For each pair of keys encountered, the appropriate distribution is used to randomly generate a time value that falls within the range as defined by the statistical model. That value is used as the delay between sending the two key events. When the driver decides to send a specific keystroke, it will immediately process it per the USB HID protocol, queuing a keyboard report as a USB interrupt transfer request that contains the key being pressed and any modifiers (shift, alt, etc.). The operating system will periodically poll the interrupt transfer pipe and retrieve the data, delivering it to any requesting applications, i.e., dprocess in our case.

2.3 Keyboard Shadowing

The operation of a decoy keyboard in parallel with a real keyboard creates an outlier configuration. Malware could simply browse all I/O devices on the

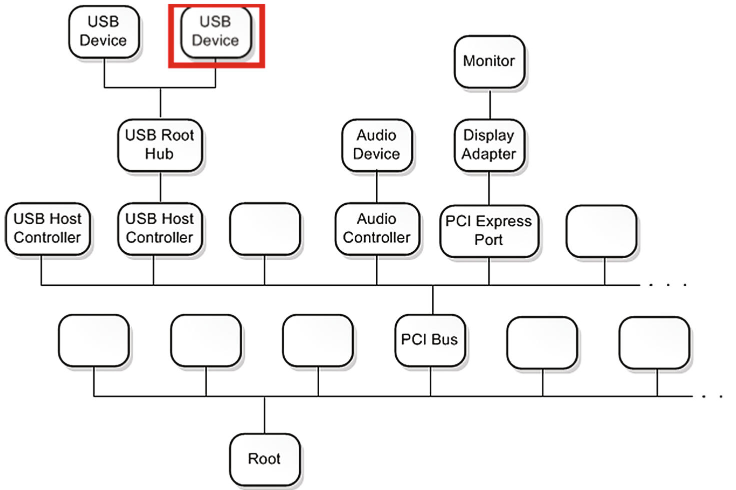

Fig. 3. Location of the decoy keyboard in the device tree. (Source: Microsoft Hardware Dev Center [42])

computer, and then check if multiple keyboards are attached to the compromised computer. Computers with two physical keyboards are not common, consequently the decoy keyboard needs to operate when the user is not typing any keys on the real keyboard. We have devised a technique, which we refer to as the keyboard shadowing mechanism (Kshadow), to detect windows of time when the user is not using the keyboard. The duration of those time windows of inactivity varies from several seconds to several minutes, and at times may last one or more hours depending on the work situation a user is in. For example, the user may be reading a document, and occasionally use the real keyboard to perform keyword searches on it. The user may be attending an hour-long presentation, in which the user mostly listens and sometimes writes down notes. The user may also be away from the computer for extended periods of time, such as when going to lunch or when participating in a meeting. Note that, since Kshadow runs in the kernel, there is now running process for it in user space.

Kshadow signals the DK driver when a time window of inactivity begins, and again when it ends. A `go` signal gives the DK driver a green light to start sending keystrokes to a dprocess. A `stop` signal notifies the DK driver that the physical keyboard is now producing keystrokes. It is time that the DK driver quickly wraps up the communication with a dprocess, and goes to sleep until the next signal from Kshadow arrives. The DK driver may choose to not use a time window of inactivity entirely, especially if the time window is long. In those cases, the DK driver selects specific portions of the time window in which to send keystrokes to a dprocess. The remaining fractions of the time window are left to be inactive. Keyboard shadowing is transparent to the user, who does not see or interact with the decoy keyboard. The user types on the physical keyboard as

if the decoy keyboard were not present on the computer. Only one keyboard is discoverable at any time on the computer, with characteristics that match those of the physical keyboard.

We now discuss the inner workings of keyboard shadowing, with reference to Fig. 4. In Windows, the I/O system is packet driven [40]. An I/O device in general is operated by a stack of drivers. The driver stack of an I/O device is an ordered list of device objects, i.e., device stack, each of which is linked with the driver object of a kernel driver. A device object is a C struct that describes and represents an I/O device to a driver, whereas a driver object is a C struct that represents the image of a driver in memory [40]. The I/O requests that read keystrokes from a keyboard are packaged into data structures called I/O request packets (IRPs) [40]. An IRP is self contained, in the sense that it contains all the data that are necessary to describe an I/O request. IRPs originate from a component of the I/O system called I/O manager, which is also responsible for enabling a driver to pass an IRP to another driver.

An IRP traverses the device stack top to bottom. It is processed along the way by the drivers in the driver stack using the I/O manager as an intermediary. Once an IRP is fully processed by those drivers and thus reaches the bottom of the driver stack, the lowest driver populates its payload with scancodes. A scancode is a byte that corresponds to a specific key on the keyboard being pressed or released. The IRP may climb back up the driver stack. At the end, the I/O manager responds to the caller thread in user space by passing the keystrokes to it. The keyboard class driver referenced in Fig. 4 does IRP processing that applies generally to all keyboards, regardless of their hardware, low-level design, and type of hardware connection to the computer. When a process in user space requests to read keystrokes, the I/O manager converts the request into an IRP and sends it to the driver located at the top of the stack. Here is how.

The I/O operation performed by an IRP is indicated by a field called major function code, which may be accompanied by a minor function code. The I/O manager accesses the device object located at the top of the stack, and from there follows a pointer to the corresponding driver object. At that point, the I/O manager uses the major function code as an index into a dispatch table of the driver object, and obtains the address of a driver routine to call. The routine belongs to the keyboard class driver, which can now process the IRP and subsequently pass it down the driver stack. All the other drivers in the stack are given an opportunity to process IRPs this way as well. Without any keyboard shadowing in place, the keyboard class driver passes the IRP to a function driver. Generally speaking, the function driver of an I/O device has the most knowledge of how the device operates. The function driver presents the interface of the device to the I/O system in the kernel.

The function driver of the physical keyboard is an HID driver, which is referred to as keyboard HID client mapper driver. It is written in an independent way from the actual transport. Possible transports can be USB, Inter-Integrated Circuit (I2C), Bluetooth, and Bluetooth Low Energy. In the case of the computers that we used for code development and research in this work, the HID

class driver serves as a bridge between the keyboard HID client mapper driver and a USB bus. The reader is referred to [20] for a thorough description of HID concepts and architecture. Keyboard shadowing is implemented as a filter driver, which is positioned between the keyboard class driver and the keyboard HID client mapper driver, as depicted in Fig. 4. Kshadow creates a filter device object (FiDO). This FiDO is similar to the functional device object (FDO) and physical device object (PDO) created by the other drivers in the stack. These device objects are only different in the type of drivers to which they represent an I/O device.

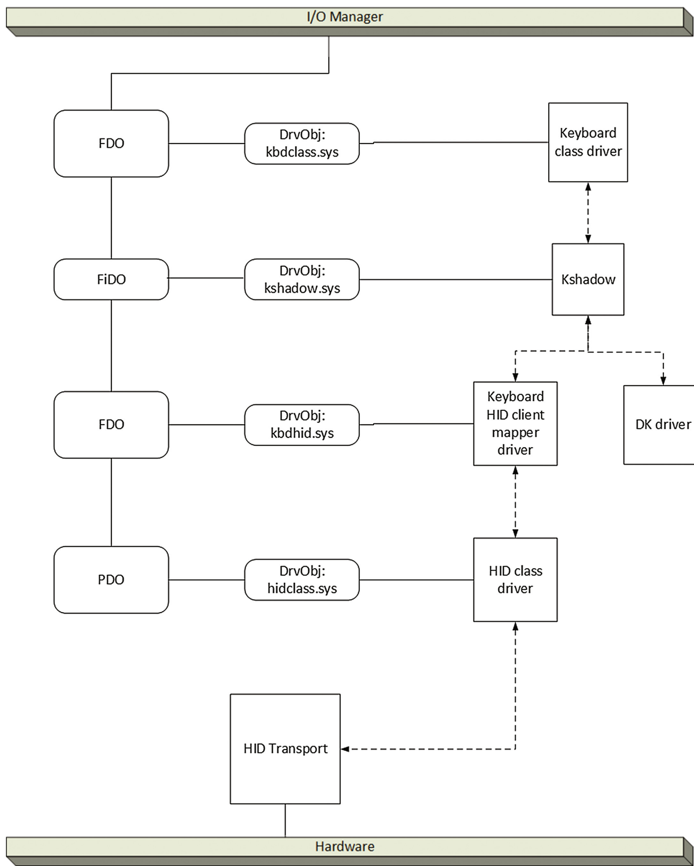

Fig. 4. Integration of keyboard shadowing with the driver stack of an HID keyboard.

By registering with the I/O manager as a filter driver, Kshadow gets to see all IRPs bound for the keyboard, physical or decoy. Kshadow has access to the process ID of dprocess. Furthermore, for each IRP, Kshadow retrieves the process ID of the thread that requested the I/O operation. This is how Kshadow

knows whether an IRP originated in dprocess or in another process. Regardless of the source, Kshadow receives IRPs from the keyboard class driver. If an IRP originated in a process other than dprocess, Kshadow passes it down to the keyboard HID client mapper driver. If an IRP originated in dprocess, Kshadow simply passes it to DK driver, which populates it with the scancodes of emulated keystrokes. Kshadow acquires high resolution time stamps to measure the interval between the current time and the time an IRP was last seen going down the driver stack. The Windows kernel provides APIs such as `KeQueryPerformanceCounter()` that are highly accurate.

If no IRPs from processes other than dprocess arrive for an interval of time that exceeds a given threshold, Kshadow marks the beginning of a time window of inactivity and thus sends a `go` signal to the DK driver. All communications between Kshadow and DK driver take place through direct function calls, and do not involve the I/O manager. This is because DK driver is not part of the driver stack, consequently no device object is created for it. It is slightly more challenging for Kshadow to spot the end of a time window of inactivity. We deem that inactivity ends when the driver at the bottom, which is also referred to as a miniport driver, completes its processing of a pending IRP. The completion event is due to the user pressing a key on the keyboard, and the miniport driver subsequently reading the scancode byte and placing it on the IRP. We leverage the fact that the completed IRP could be made to climb up the driver stack.

When Kshadow notices a completed IRP coming from underneath, it knows that the physical keyboard has become active and thus sends a `stop` signal to the DK driver. However, a completed IRP does not just climb up the driver stack by itself. Kshadow registers one of its functions as an `IoCompletion` routine for an IRP, before passing it down the driver stack. When the IRP is complete, the `IoCompletion` routines of all higher-level drivers are called in order. When the `IoCompletion` routine of Kshadow is invoked, Kshadow marks the end of the time windows of inactivity and DK driver stops sending emulated keystrokes to dprocess.

2.4 First and Second Order Detection

First order detection of a keylogger happens when the malware attempts to intercept keystrokes and is detected in the act. Second order detection refers to detection at a time that postdates the keystroke interception mounted by a keylogger. As we discuss later on in this paper, our approach can attain first order detection of various forms of keyloggers. Nevertheless, its main strength is in second order detection of keyloggers. When we started this research, our goal was to deliver effective second order detection. First order detection is primarily consequential, and was noticed mostly during the practical tests against live malware. We base this work on a simple observation: keyloggers intercept data for attackers to use. Of course, not all intercepted data will be used, but some of those data will. In fact, malicious use of intercepted data is the reason behind the very existence of keyloggers.

The idea that underpins this work is to use a decoy keyboard to generate emulated and hence decoy keystrokes for keyloggers to intercept. Keyloggers commonly encrypt the data they intercept on a compromised computer, and then send those data to the attacker over the network. Furthermore, keyloggers commonly come as one of the modules of larger malware. Other modules open up backdoors into the compromised computer, and enable the attacker to access any resources. Yet other modules intercept filesystem traffic, spy on a user over the webcam, or authenticate to other computers using stolen credentials. The malware are detected when they use decoy data leaked by the decoy keyboard. In our prior work we have created decoy network services, which require authentication. In this case, we need to leak an account by using the decoy keyboard to make it appear as if a user is typing his or her username and password to authenticate to those services.

More specifically, we have explored a decoy network interface card (DNIC), which makes a computer appear to have access to an internal network [41]. Neither the DNIC nor the internal network exist for real. Furthermore, they are both masked to the user, consequently the user does not see and hence does not access them. We have developed an Object Linking and Embedding (OLE) for Process Control (OPC) server, which provides power grid data after authenticating the client. All of these decoys are implemented in the kernel, therefore no network packet ever leaves the computer for real. The decoy keyboard leaks an OPC server account, which includes the IP address of the server computer, the name of the OPC server, and of course a username and password. The IP address in question is reachable over the imaginary network. Once malware intercepts the OPC server account from the decoy keyboard and thus uses it to access the OPC server over the DNIC, we attain second order detection.

Another example involves the leakage of credentials that provide access to a virtual private network (VPN), which is reachable only over the DNIC. VPNs are of particular interest to attackers, since they commonly lead to protected resources. For instance, the BlackEnergy malware that compromised a regional power distribution company in Ukraine was able to access electrical substation networks through a VPN using stolen VPN credentials. We can display an imaginary VPN over the DNIC. The imaginary VPN is accessible via credentials leaked by the decoy keyboard, leading to an unequivocal second order detection. It is of paramount importance to this work that the attacker does not discard the data that the keylogger had intercepted from the decoy keyboard. This is why the model of human keystroke dynamics, which we discussed earlier in this paper, is so critical to this work. If we can leak data in a way that resembles those produced by a real keyboard, we give the attacker no reason not to use the decoy data. For consistency reasons, the decoy keyboard can provide decoy non-sensitive data for other decoy processes. For example, the decoy keyboard can fill a webform through a decoy web browser process.

3 Evaluation

The decoy keyboard approach was installed and run on a 64-bit Windows 10 virtual machine in our lab. The lab was isolated both logically and physically from any physical computer networks. Firstly, we tested the ability of the decoy keyboard to co-exist with a real keyboard. We simply used a computer equipped with a decoy keyboard for several days to do our usual work. We paid attention to all details related to the keyboard use. We observed no visible delays when typing on the physical keyboard. There were numerous periods of keyboard inactivity, in which we were away from the computer. The logs show that the decoy keyboard had been in operation multiple times, however we saw no anomalies with the use of the real keyboard when we returned to work on the computer. The only observable was that the screen saver and the power saving mode appeared to be affected by the operation of the decoy keyboard. As a dprocess requests keystrokes, and the DK driver generates them, normal computer activity is created, consequently Windows assumes that there is a user working on the computer.

The delay in IRP processing caused by Kshadow, as the user types on the physical keyboard, is characterized by the data of Fig. 5. The lower chart indicates the delays that occur when Kshadow passes the IRPs down the driver stack as soon as they arrive. This graph is given solely for comparison reasons. The higher graph shows the delays when Kshadow works to determine if a time window of inactivity has begun. The greater delays are due to Kshadow registering one of its functions as an `IoCompletion` routine, acquiring high-resolution time stamps to measure time intervals, and retrieving the process ID for the thread that originally requested to read a keystroke. Other overhead is due to starting and maintaining a kernel thread.

The delays caused by the overall approach when a user returns to his or her workplace and presses a key on the keyboard are given in Fig. 6. This is the situation in which the decoy keyboard has been operating for some time, and now the user needs the physical keyboard back in operation. The data pertain to 10 separate occurrences of the aforementioned situation. The lower chart shows the delays that occur when the DK driver is able to respond to the `stop` signal immediately. This only happens when the DK driver is not emulating keystrokes, and dprocess does not exist or is suspended and hence is not reading keystrokes. Otherwise the DK driver needs some time to wrap up its keystroke emulation, halt dprocess, and altogether get out of the way. The duration of these finalizations depends on the data that the decoy keyboard is leaking through the attack surface and hence may vary. There is also some delay occurring when the keyboard transitions in the opposite direction, namely from physical to decoy. Nevertheless, the user is not affected, since he or she is not typing on the physical keyboard at that time.

We examined the arrival times of keystrokes in user space for the purpose of testing the statistical model. Without incorporating the model in the DK driver, keystrokes arrive at an almost constant rate. The delay between any two keys is the same, and quite minimal. Clearly this is an obvious indication that the

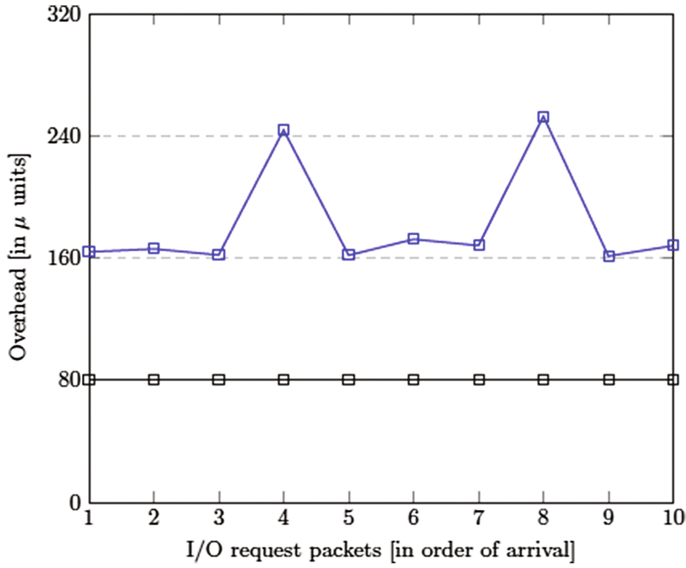

Fig. 5. I/O filtering overhead while the physical keyboard is in use.

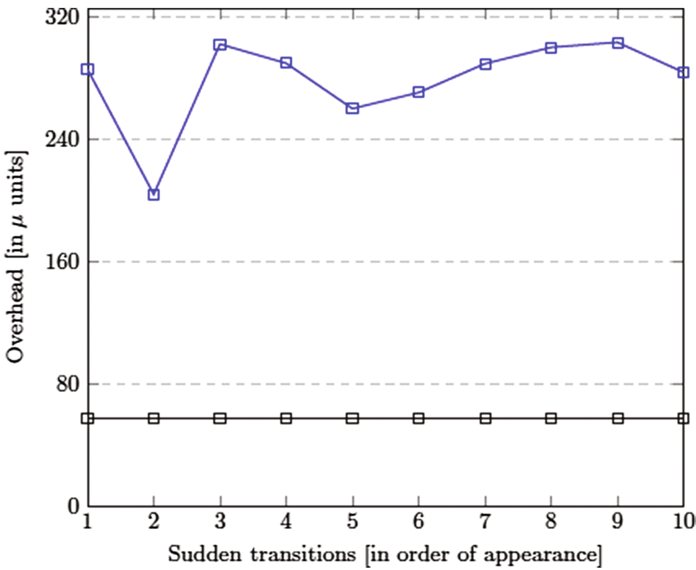

Fig. 6. Decoy-to-physical transition overhead.

keyboard is a decoy. With the statistical model in use within the DK driver, the keystrokes arrive at times that match the typing profile of an actual user. We have not included the corresponding charts due to room limitations.

We now discuss the evaluation of the effectiveness of our approach. We tested the decoy keyboard against live malware. The malware corpus was comprised of samples that were obtained from public malware repositories, namely Open Malware, AVCaesar, Kafeine, Contagio, StopMalvertising, and unixfreaxjp. Those repositories provide malware samples for academic research. The malware samples were known, and included remote access tools (RATs), worms and trojans, and also viruses. We used the IDA Pro tool to analyze the malware samples to the greatest degree that we could in order to identify and remove goodware. We also did analysis work to identify malware samples that are of the same malware, but appear different because of polymorphic or metamorphic techniques. At the end, the malware corpus consisted of 50 malware samples. Most of those malware samples have a history of involvement in malware campaigns in the recent years, therefore are valid and pertinent for testing purposes.

Some of the malware samples use keylogger modules that intercept keystrokes by probing their target keyboard. Those probes resulted in IRPs bound for the target keyboard. When the decoy keyboard was in operation, the malicious IRPs reached Kshadow and DK driver, causing a first order detection of the malware. The other malware samples that escaped first order detection needed to be tested against second order detection. In that regard, we did not operate the decoy keyboard against real human attackers on the Internet. We are a University and thus are not in the position to run real-world cyber operations at this time. Nevertheless, we tested whether or not the decoy keystrokes were intercepted by the malware, and the malware subsequently attempted to send those keystrokes to attackers over the network. If we succeeded in causing malware to attempt to send decoy data to attackers, then the use of those decoy data by the attackers, and hence a second order detection of the malware by our approach, is only a matter of time.

Our testing differed depending on the type of malware. Let us first discuss the case of RATs. We started with testing the decoy keyboard approach against QuasarRAT and jRAT, and later realized that they resemble the software architecture and interception tactics of several other RATs such as Bandook, Ozone, Poisonivy, and njRAT. These RATs are comprised mainly of two executables, a client and a server. The server executable is run on a compromised computer, whereas the client executable is run on the attacker's machine. The server executable performs keylogging, and then sends the data to the client. The client executable typically presents a graphical user interface (GUI) panel to the attacker, which the attacker uses to select target computers, as well as the operations to perform on them. A typical control panel is shown in Fig. 7. The keylogger module is selected.

As the decoy keyboard found several time windows of inactivity, it sent keystrokes to dprocess. The keystrokes were intercepted by the server executable, which sent them to the client executable. The client executable in turn displayed

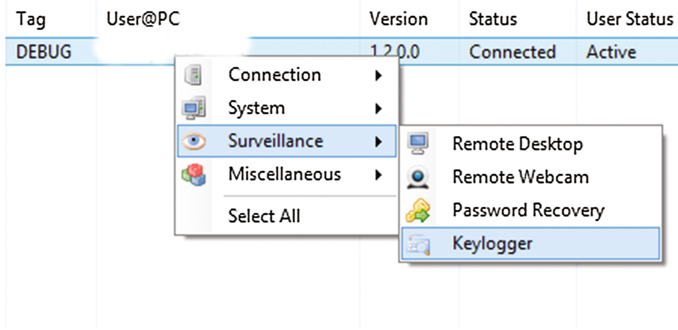

Fig. 7. The control panel of QuasarRAT.

the intercepted keystrokes on the GUI panel. The decoy data that were leaked by the decoy keyboard were now listed on the GUI panel. In the case of the other malware samples, the testing was challenging as there were no GUI panels to display the intercepted data. In those cases, we used a kernel debugger to set breakpoints on instructions of the malware that accessed intercepted keystrokes. We dumped the intercepted data on the debugger's console, and verified that some of those data were indeed data generated by the DK driver.

In conclusion, all of the samples in the malware corpus intercepted keystrokes generated by the decoy keyboard, and either sent them to the client executable running on a computer on the local network, or attempted to send them to external IP addresses. In the latter case, the network packets were eventually dropped due to the testbed isolation. There were no false alarms raised, because no goodware ever requested to read keystrokes after the user went away from the computer and a decoy keyboard was made active.

4 Related Work

Decoy I/O devices are intrinsically a form of deception, and there has been a great deal of prior work in the field. Cohen's Deception Toolkit [9] in 1997 was one of the first publicly available programs designed to deceive and identify attackers by presenting fake information and vulnerabilities [10]. Honeypots and honeyfiles, decoy machines and files designed to entice and trap attackers, have also been explored quite heavily. In 1989, Stoll published a book that detailed some of the earliest uses of honeypots and honeyfiles as he worked to catch a hacker that infiltrated the computer systems at Lawrence Berkley National Laboratory [11]. By Presenting Attractive yet fake files to the attacker, he was able to catch them "in the act," eventually leading to their arrest. These concepts have since been applied towards attracting automated software instead of real people, but the underlying idea remains the same.

A decoy I/O device as presented in this research has two main differences from existing deception tactics and decoys such as honeypots and honeyfiles.

The first is that it can (and is intended to) be deployed on an active, production machine, running in the background transparent to the user. The second is that it is specifically designed to emulate a real I/O device down to the hardware level, making it as indistinguishable from a real device as the complexity of the decoy communications allow. A typical honeypot, such as Sebek [12], is a system designed solely for logging and reporting attacks that presents itself as a potential target through the activity of a typical system in production use. However, due to the nature of the design, the honeypot itself cannot be in active use; any incoming connections are either malicious in nature or from someone who has stumbled upon it by accident.

Sebek and other high-interaction honeypots will imitate a complete machine, and thus require the full resources of one, and are expensive to maintain. Low-interaction honeypots like PhoneyC [13] are designed to only present certain vulnerable services, and require less resources to operate. In order to operate multiple honeypots on one physical computer, virtual machines can be utilized, but malware such as Conficker [14] can detect that they are being run virtually and change their behavior, or simply avoid the honeypot altogether. Along that vein, Rowe proposed using fake honeypots as a defensive tactic by making a standard production machine appear to be a honeypot [15], scaring away any would-be attackers by emulating the signatures and anomalies that are typically associated with honeypots.

Anagnostakis et al. proposed a hybrid technique called a "shadow honeypot" [16] that utilizes actual production applications with embedded honeypot code. Incoming connections are first filtered through an anomaly detection process, which will redirect possible malicious traffic to the shadow honeypot, and normal traffic to the standard server. However, the shadow honeypot is running the same server application with the same internal state, with the added honeypot code serving to analyze the behavior of the request. The shadow honeypot can then send valid requests to the real server, and any change in state from malicious activity is thrown out. Spitzner discussed the issue of insider threats [17], and how they target actual information that they can use, thus the honeypot could provide fake data that appears attractive. This data could be business documents, passwords, and so on, and would be a "honeytoken" designed to redirect the attacker to the honeypot. This is similar to the research presented here in that fake data attracts an attacker to the decoy I/O device, but is on an entirely different scale and still requires a high-interaction honeypot to implement.

The honeytokens proposed by Spitzner are a type of honeyfile, a concept explored by Yuill et al. [18] which are commonly used on honeypots but have also been placed on systems in actual use. Software can then analyze traffic to and from the files with the assumption that any activity is malicious in nature, as those files are not in use by legitimate applications. This concept of a Canary File, or a honeyfile placed amongst real files on a system, was proposed by Whitman [19], and he also discusses the automatic generation and management of the Canary Files. However, the content of automatically generated files is

difficult to present as authentic upon inspection, and malware that resides at the operating system level is able to examine file access patterns to ignore files that are not being used.

5 Conclusion

The decoy keyboard is a novel anti-malware approach that is transparent to both normal users and attackers. The approach requires no prior knowledge of malware code and data structures to be able to detect them, and can operate on computers in production. The filter driver is able to shadow a physical keyboard. It can reliably guide a low-level deceptive driver in the generation of decoy keystrokes for malware to intercept. The decoy keystrokes are delivered to malware according to a timing model that makes both the decoy data and their delivery quite consistent with the behavior of a physical keyboard. The decoy keyboard is safe to operate, and does not interfere with the user's work. The decoy keyboard showed to be effective against a large malware corpus, attaining approximated second order detections and some first order detections as well.

Acknowledgments. This material is based on research sponsored by U.S. Air Force Academy under agreement number FA7000-16-2-0002. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of U.S. Air Force Academy or the U.S. Government.

Appendix A: Malware Detection

Current malware detection (including keyloggers) is based on both static and dynamic analysis. Malware detection through static analysis consists of scanning an executable file for specific strings or instruction sequences that are unique to a specific malware sample [1,2,21]. This is a primary type of detection employed by commercial anti-virus software packages; in addition to simply scanning files present on a computer, the keyboard and other device stacks are inspected and interceptions by known malware are flagged and reported, but lists of known signatures must be kept and frequently updated. The main limitation of the static analysis techniques is that malware can change its appearance by means of polymorphism, metamorphism, and code obfuscation. Other static analysis research focused on higher-order properties of executable files, such as the distribution of character n-grams [22,23], control flow graphs [3,24], semantic characteristics [4], and function recognition [25–27]. Dynamic analysis is another field of ongoing research that studies the execution flow of a malware binary using methods such as function call monitoring and information flow tracking [5]. Tools like Detours [6] allow an analyst to hook into the function calls of a piece of malware and execute their own code for investigative purposes before returning control

to the original program. This type of hook can be performed on binary files located on disk as well as by modifying the memory space of a currently running process. Using these techniques, researchers are able to learn various indicators, which are then used to detect that malware. Detection indicators include disk access patterns [28], malspecs [29], sequences of system calls and system call parameters [30], and behavior graphs [31].

All of these methods are rooted in and constrained by the concept of analyzing existing malware. The main advantage and differentiator of the decoy keyboard approach over such a large body of malware detection research is that a decoy keyboard does not require any prior analysis of a malware sample in order to detect it. A decoy keyboard can detect malware encountered for the very first time, whereas the techniques from the mentioned large body of malware detection research require that an instance of malware be given to them as input for analysis. A specific instance of the malware needs to be detected by other means – someone has to provide the malicious code, along with an explicit and validated statement that it is malware. Those techniques will then be used to analyze the program, which at that time is known to be malware, and thus will learn indicators such as those discussed in the previous paragraph. Those indicators are subsequently used to detect other instances of the malware. This introduces a period of time between the release and discovery of new malware and the update of any software to protect against it during which targeted systems are vulnerable. Additionally, kernel-based malware has such a degree of access to the system and underlying processes that it may very well be able to find and circumvent any installed security software.

Some work has been done to counteract the deficiencies of static and dynamic analysis by using machine learning techniques such as behavioral clustering [7] and classification [8], but the accuracy of these techniques is dependent on the quality of the training set; malware that is significantly different in operation from the majority may still manage to evade detection.

References

1. Christodeorescu, M., Jha, S.: Static Analysis of Executables to Detect Malicious Patterns. Department of Computer Sciences, Wisconsin Univ-Madison (2006)
2. Griffin, K., Schneider, S., Hu, X., Chiueh, T.C.: Automatic generation of string signatures for malware detection. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 101–120. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04342-0_6
3. Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Büschkes, R., Laskov, P. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. LNCS, vol. 4064, pp. 129–143. Springer, Heidelberg (2006)
4. Feng, Y., Anand, S., Dillig, I., Aiken, A.: Apposcopy: semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 576–587 (2014)

5. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 6 (2012)
6. Hunt, G., Brubacher, D.: Detours: binary interception of Win32 functions. In: 3rd Usenix Windows NT Symposium (1999)
7. Bailey, M., Oberheide, J., Andersen, J., Mao, Z.M., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 178–197. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74320-0_10
8. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011)
9. Cohen, F.: The deception toolkit. Risks Digest, vol. 19 (1998)
10. Cohen, F.: A note on the role of deception in information protection. Comput. Secur. **17**(6), 483–506 (1998)
11. Stoll, C.: The Cuckoo's Egg: Tracing a Spy through the Maze of Computer Espionage. Doubleday, New York (1989)
12. Balas, E.: Know your enemy: Sebek. The Honeynet Project (2003)
13. Nazario, J.: PhoneyC: A Virtual Client Honeypot. LEET, vol. 9, pp. 911–919 (2009)
14. Leder, F., Werner, T.: Know your enemy: Containing conficker. The Honeynet Project (2009)
15. Rowe, N.C., Duong, B.T., Custy, E.J.: Defending cyberspace with fake honeypots. J. Comput. 2(2) (2007)
16. Anagnostakis, K.G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E.P., Keromytis, A.D.: Detecting targeted attacks using shadow honeypots. In: Usenix Security (2005)
17. Spitzner, L.: Honeypots: catching the insider threat. In: 19th Annual Computer Security Applications Conference, pp. 170–179 (2003)
18. Yuill, J., Zappe, M., Denning, D., Feer, F.: Honeyfiles: deceptive files for intrusion detection. In: Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, pp. 116–122 (2004)
19. Whitham, B.: Canary files: generating fake files to detect critical data loss from complex computer networks. In: The Second International Conference on Cyber Security, Cyber Peacefare and Digital Forensic, pp. 170–179 (2013)
20. Microsoft Device and Driver Technologies: HID drivers (2016). https://msdn.microsoft.com/en-us/windows/hardware/drivers/hid/index
21. Szor, P.: The Art of Computer Virus Research and Defense. Pearson Education, Indianapolis (2005)
22. Li, W.-J., Stolfo, S., Stavrou, A., Androulaki, E., Keromytis, A.D.: A study of malcode-bearing documents. In: Hämmerli, B., Sommer, R. (eds.) DIMVA 2007. LNCS, vol. 4579, pp. 231–250. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73614-1_14
23. Li, W.J., Wang, K., Stolfo, S.J., Herzog, B.: Fileprints: identifying file types by n-gram analysis. In: Information Assurance Workshop, Proceedings from the Sixth Annual IEEE SMC 2005, pp. 64–71 (2005)
24. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: International Workshop on Recent Advances in Intrusion Detection, pp. 207–226 (2005)
25. Kruegel, C., Robertson, W., Vigna, G.: Detecting kernel-level rootkits through binary analysis. In: 20th Annual Computer Security Applications Conference, pp. 91–100 (2004)

26. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 174–187 (2005)
27. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy, pp. 32–46 (2005)
28. Felt, A., Paul, N., Evans, D., Gurumurthi, S.: Disk level malware detection. In: Poster: 15th Usenix Security Symposium (2006)
29. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 1st India Software Engineering Conference, pp. 5–14 (2008)
30. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 122–132 (2012)
31. Kolbitsch, C., Comparetti, P.M., Kruegel, C., Kirda, E., Zhou, X.Y., Wang, X.: Effective and efficient malware detection at the end host. In: USENIX Security Symposium, pp. 351–366 (2009)
32. Umphress, D., Williams, G.: Identity verification through keyboard characteristics. Int. J. Man Mach. Stud. **23**(3), 263–273 (1985)
33. Gaines, R.S., Lisowski, W., Press, S.J., Shapiro, N.: Authentication by keystroke timing: Some preliminary results. No. RAND-R-2526-NSF. RAND CORP (1980)
34. KeyTrac Keyboard Biometrics. http://www.keytrac.net
35. Banerjee, S.P., Woodard, D.L.: Biometric authentication and identification using keystroke dynamics: a survey. J. Pattern Recogn. Res. **7**(1), 116–139 (2012)
36. Roth, J., Liu, X., Metaxas, D.: On continuous user authentication via typing behavior. IEEE Trans. Image Process. **23**(10), 4611–4624 (2014)
37. Feit, A.M., Weir, D., Oulasvirta, A.: How we type: movement strategies and performance in everyday typing. In: Proceedings of the 2016 Chi Conference on Human Factors in Computing Systems, pp. 4262–4273 (2016)
38. Choi, Y.: Keystroke patterns as prosody in digital writings: a case study with deceptive reviews and essays. In: Empirical Methods on Natural Language Processing, p. 6 (2014)
39. Leys, C., Ley, C., Klein, O., Bernard, P., Licata, L.: Detecting outliers: do not use standard deviation around the mean, use absolute deviation around the median. J. Exp. Soc. Psychol. **49**(4), 764–766 (2013)
40. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows Internals, Part 1 and 2, 6th edn. Microsoft Press, Redmond (2012)
41. Rrushi, J.: NIC displays to thwart malware attacks mounted from within the OS. Comput. Secur. **61**(C), 59–71 (2016)
42. Microsoft Hardware Dev Center: Device nodes and device stacks. https://msdn.microsoft.com/en-us/windows/hardware/drivers/gettingstarted/device-nodes-and-device-stacks
43. Newton, D.: Virtual Multiple HID Driver (multitouch, mouse, digitizer, keyboard, joystick). https://github.com/djpnewton/vmulti