COMPUTERS & SECURITY 77 (2018) 209–225

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Automated analysis of freeware installers promoted by download portals

Alberto Geniola [a], Markku Antikainen [b,*], Tuomas Aura [a]

[a] Aalto University, Finland
[b] Helsinki Institute for Information Technology, University of Helsinki, Finland

ARTICLE INFO

Article history:
Received 24 October 2017
Accepted 26 March 2018
Available online 31 March 2018

Keywords:
Potentially-unwanted program
Pay-per-install
UI-automation
Man-in-the-middle Malware

ABSTRACT

We present an analysis system for studying Windows application installers. The analysis system is fully automated from installer download to execution and data collection. The system emulates the behavior of a lazy user who wants to finish the installation dialogs with the default options and with as few clicks as possible. The UI automation makes use of image recognition techniques and heuristics. During the installation, the system collects data about the system modification and network access. The analysis system is scalable and can run on bare-metal hosts as well as in a data center. We use the system to analyze 792 freeware application installers obtained from popular download portals. In particular, we measure how many of them drop potentially unwanted programs (PUP) such as browser plugins or make other unwanted system modifications. We discover that most installers that download executable files over the network are vulnerable to man-in-the-middle attacks. We also find, that while popular download portals are not used for blatant malware distribution, nearly 10% of the analyzed installers come with a third-party browser or a browser extension.

1. Introduction

Most computer users download and install some freeware applications from the Internet. The source is often one of the many download portals, which aggregate software packages and also offer locations for hosting them. It is common concern that the downloaded software might be infected with malware or have other unwanted side effects. Freeware installers are also known for dropping of potentially unwanted programs (PUP) to the user's computer. PUP and other unwanted system modifications to desktop computers can also be considered a security threat (Emm et al., 2016; Wood et al., 2016). This phenomenon is partly caused by the *pay-per install* (PPI) business model where freeware software developers can monetize their software effectively by bundling it with other third-party applications or by promoting some software and services by changing the user's default settings. This business model is not always illegal as the application installer may inform the user about the third-party software and even allow her to opt-out from installing third-party applications. However, this is often done in a way that the user is not completely aware of the choices he makes.

In this paper, we set out to analyze how prevalent are the security and PUP problems among the software obtained from

* Corresponding author.
E-mail addresses: alberto.geniola@studenti.polito.it (A. Geniola), markku.antikainen@helsinki.fi (M. Antikainen), tuomas.aura@aalto.fi (T. Aura).
https://doi.org/10.1016/j.cose.2018.03.010

download portals. For this, we create an automated analysis system that downloads and installs the applications in a sandbox while monitoring the target system. The sandbox emulates the behavior of a lazy user who tries to complete the installation process with the default settings of the installer. It does this with the help of image recognition on screenshots and heuristic rules. During the whole process, we record network traffic and modifications to the target system. We demonstrate the capabilities of the system by analyzing nearly 800 popular software installers from eight different download portals.

As hinted, we have two distinct goals. First, we create a scalable and fully automated tool for analyzing a large number of application installers. Unlike other existing application analysis sandboxes (e.g. Cuckoo Sandbox by Guarnieri et al., 2012), our tool is not only a plain sandbox but can also interact with application installers. Our second goal is to use the system to analyze large quantities of software from different download portals in order to better understand the prevalence of any security problems in them. Unlike earlier research on PPI and PUP, such as those presented by Caballero et al. (2011) and Thomas et al. (2016), we do not try to differentiate between legitimate and malicious actions but try to cover all potentially unwanted changes to the system. This not only gives insight to the prevalence of any problems but also teaches up about the software installers and download portals in general.

More specifically, our contributions are the following:

- We create a scalable, fully automated, sandboxed analysis system for software installers. The system uses UI automation to emulate user interaction and monitors the installation process. The system supports virtualized as well as bare-metal sandboxes. The system has been published as open source.[1]
- To show the capabilities of the system, we use it to analyze 792 popular freeware installers crawled from eight popular download portals. The analysis covers file system access, registry modifications, and network traffic. We look for indications of unwanted software drops, other potentially unwanted changes to the system, and vulnerabilities in the network communication of the installers. Our main findings include that while the download portals do not distribute malware, 1.3 % of the installers led to the installation of a well-known potentially unwanted application (PUP) and nearly 10 % of the installers came with a third-party browser (e.g. Chrome) or a browser extension. Furthermore, we found that the installers often download the application binaries over HTTP and that over half of these are do not verify the integrity of the binary and are thus vulnerable to man-in-the-middle (MitM) attacks. While some of the analysis results have been published earlier (Geniola et al., 2017), the results and discussion presented in this paper are more comprehensive than what has been published earlier.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the overall architecture of the analysis system. Section 4 explains how we were able to automatically interact with the UI's of the installers. Section 5 moves towards using the analysis system and describes how the system was used to analyze a large number of freeware installers. Analysis results are presented in Section 6 and further discussed in Section 7. Section 8 concludes the paper.

2. Background

This section describes the related work and ideas on which our research is based.

2.1. *Potentially unwanted programs*

Downloading applications from the Internet can be dangerous, and this also applies to download portals (Heddings, 2014; 2015). The applications might come with unwanted features that range from clearly malicious, such as bundled malware and spyware, to minor nuisances like changing the browser's default search engine. Such software is often referred to as *potentially unwanted programs* (PUP)[2]. We use the broad definition of Goretsky (2011), which states that a PUP is an application or a part of an application that installs additional unwanted software, changes the behavior of the device, or perform other kinds of activities that the user has not approved or does not expect. PUP often functions in a legal and moral gray area. The threat of legal action from PUP authors has been suggested as the reason why antimalware labels it as "potentially unwanted" rather than "malicious" (Boldt and Carlsson, 2006; McFedries, 2005).

Recently, Kotzias et al. (2016) have shown that freeware installers only rarely come bundled with critical malware. More often, the system modifications are just unnecessary and unexpected. The user may even be informed about them, for example, in the end-user licence agreement (EULA), and the installer may allow a careful user to opt out of unwanted features. However, as pointed out by Böhme and Köpsell (2010); Motiee et al. (2010), users do not always read EULAs and may be habituated to accept default settings and ok any warnings. This *rushing-user* behavior leads the user to giving *uninformed consent* to the system modifications. Moreover, PUP installers often come with a complex EULAs (Good et al., 2005), which users are more likely to accept blindly (Bruce, 2005). Solutions to this problem have been proposed (Boldt and Carlsson, 2006). For example, Boldt et al. (2008) showed that it is possible to detect some classes of spyware can be detected by analyzing the EULAs. However, none of the proposals has been widely adopted.

On mobile platforms, the problem of uninformed consent has been solved so that the operating system informs the user about the permissions given to each application. This may happen either at the install time (e.g. Android 5 and earlier), or when the application requests access to restricted

[1] The analysis system is available at https://github.com/albertogeniola/TheThing/tree/master.

[2] Potentially Unwanted Application (PUA) is another often used term.

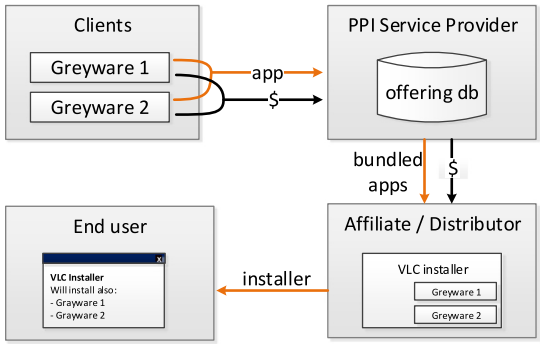


Fig. 1 – PPI business model.

resources (e.g. Android 6). This, however, requires that all inter-application communication go through a policy enforcement point. Because of the historical baggage of desktop operating systems, the same solution is not currently feasible on them.

As a summary, while the anecdotal evidence showing that download portals distribute PUPs is plentiful (see e.g. Slade, 2015), the true extent of this problem has not been studied methodically. We aim to fill this gap by providing a comprehensive analysis of nearly 800 application installers retrieved from the most popular download portals. Because manual analysis of such number of installers is impractical, we have to create an automated analysis system for this purpose. While the PUP phenomenon is not limited to a single operating system or platform, we focus purely on Microsoft Windows, which is the most popular operating system on desktop and laptop computers with 84% market share according to Statcounter (2017).

2.2. *Pay-per-install business*

One root cause for the problem of unwanted software is the pay-per-install (PPI) business model. PPI is a monetization scheme where a software developer or distributor gets payed for dropping unrelated third-party applications to the target computer. This may be done with or without the user's consent.

As Caballero et al. (2011) describe, there are three main actors in the PPI market: *clients*, *PPI service providers* and *affiliates*. Clients are, for example, adware vendors who want to install their software on a large number of hosts. They pay a commission to the PPI service provider based on the number of successful installations. The PPI service providers are the orchestration points in this distribution model. They charge clients for every successful software installation and rely on affiliates to perform those installations. Moreover, PPI service providers implement affiliate recruitment through advertisements, so that their network reaches a larger number of affiliates. Lastly, the affiliates are the publishers and creators of popular applications. They wrap the application with third party software provided by the PPI service provider. The affiliates are then payed by the PPI service provider based on the number of third-party application installation they perform. The business model is illustrated in Fig. 1.

The PPI application installer typically downloads the third-party software from a PPI distributor. Caballero et al. (2011) reverse engineered protocols used by PPI distributors and found that the choice of applications depends on the target computer's geolocation. Another interesting result, by Kotzias et al. (2016), is that while PPI distributors do spread some known malware, this is not a very prevalent phenomenon, probably because black-listing by anti-virus vendors would hurt the PPI business.

Thomas et al. (2016) make the distinction between black-market PPI, which installs third-party applications silently in the background, and commercial PPI, which does not try to hide the installation but rather takes advantage of the rushing user behavior. In this paper, we consider commercial PPI as well as also analyze other unwanted side effects of the installers even if not part of the PPI business.

2.3. *Related work on dynamic analysis*

Unattended analysis of large numbers of binaries has been widely addressed by previous research. Some examples include Cuckoo Sandbox (Guarnieri et al., 2012), GFISandbox (Willems et al., 2007), Drakvuf (Lengyel et al., 2014) and BareCloud (Kirat et al., 2014). These sandboxes are mostly aimed at analyzing malware samples and are able to collect data on the behavior of the executed binary. Most of these employ virtualization, while BareCloud also supports bare-metal analysis.

As mentioned, these sandboxes are primarily meant for the analysis of malicious software. Thus, they typically prioritize stealthiness in their design. Also, from the aforementioned sandboxes, only Cuckoo Sandbox provides scripts for automating user interaction. However, the scripts are rather rudimentary and cannot deal with complex user interfaces. Since our study focuses on the large-scale study of legitimate (yet unwanted) applications that require complex UI automation, we opted for creating our own analysis system.

3. Automated application-installer analysis system

This section describes the analysis process and the system architecture of the analysis system. We have made the source code of the analysis system[3] and its documentation[4] publicly available.

3.1. *Methodology overview*

Our goal is to implement automated analysis of large numbers of Windows freeware installers. For this, we need a fully automated infrastructure that automatically downloads, executes and analyzes the application installers. From the beginning, we decided not to do manual work and focus on improving the automation.

On a high lever, the analysis system (1) Crawls selected download portals for Windows freeware intallers, (2) automatically runs them in guest machines with emulated user interaction, (3) monitors the modifications made to the guest

[3] https://github.com/albertogeniola/TheThing/tree/master.
[4] http://thething.readthedocs.io/en/latest/.

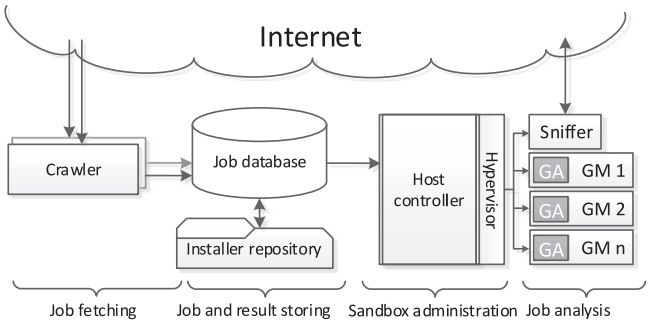


Fig. 2 – Analysis system architecture.

machine as well as network communication, and (4) saves the results for later use.

In addition to targeting a large number of installers, iterative development of the analysis system means that we needed to run thousands of installers daily. Thus, the infrastructure must be scalable and flexible. For scalability, we rely on a distributed architecture and take advantage of virtualization technologies. The amount of data collected is potentially huge, and we had to make tradeoffs to keep it under control. The whole system is designed to be modular and to run on multiple host platforms, with possibility of adding support for further guest OSs.

No existing system fulfilled all the requirements, and we opted for creating one ourselves. In particular, no existing analysis system provides tools for the complex graphical UI interactions required for emulating user behavior with unknown software. Since the goal is to analyze legitimate software installers and not viruses or worms, stealthy monitoring is not our priority.

3.2. *Analysis system architecture*

The architecture of the analysis system is shown in Fig. 2. First of all, we implemented *crawlers* for selected download portals. They retrieve installers from each site and store fetched binaries together with metadata to a database. The *job database* is a central point in the architecture and will also store the analysis results. The *host controller* is responsible for orchestrating the actual analysis. It handles the life cycle of the *guest machines* (GM in Fig. 2), in which the installers are executed. This essentially means that the host controller is responsible for (1) fetching a job from the database, (2) initializing a guest machine and serving it the installer binary, (3) pre-processing and storing data about the installation process, and finally (4) cleaning up the guest machine. Our implementation supports both virtual guest machines (as in the figure) and bare-metal guest machines. In each guest machine, there is a *guest agent* (GA in Fig. 2) that receives the installer from the host controller and drives its execution by launching it and interacting with its UI. The agent also monitors the system for modifications and reports these to the host controller. The network traffic to the guest machines is routed through a network sniffer, which captures it.

The analysis system is modular and can support any guest OS. However, we only implemented the guest agent for 32-bit Windows 7 guest machines (64-bit support is being developed at the time of writing). The other parts of the architecture are OS agnostic. The host controller and crawler were written in Python, and the platform-dependent parts were isolated so that the same architecture can be run on different hypervisors. We implemented support for Virtualbox hypervisors, Openstack cloud, and even a pool of bare metal machines. Fig. 3 illustrates some of the possible deployment strategies. In Fig. 3a, the whole analysis system runs on a singe (local) server, which natively runs the crawlers, database, and host controller. The guest machines are run as VirtualBox virtual machines on the same physical host. In Fig. 3b, the virtual machines are moved to OpenStack cloud. This enable easy scaling of the analysis system. In the third option (Fig. 3c), each guest machine is run on a dedicated physical computer. This makes the analysis of more evasive malware easier.

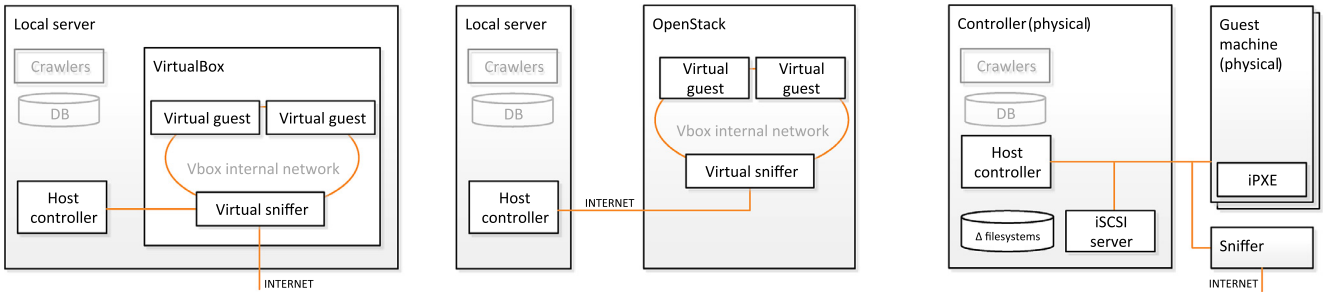
It should be noted that, because the architecture allows analyzing binaries both in virtualized as well in bare-mental guest machines, it is also possible to analyze the same binary in different environments. This could potentially be used to automate the analysis of evasive malware as any differences in the analysis results could be caused by some evasion techniques.

3.3. *Guest machine life-cycle management*

Once the database is filled with binaries, the host controller may start the guest machine. Before this, however, the guest machine needs to be initialized. This, essentially, means booting the guest machine from a *base image*, which is a disk that contains Windows 7 as well as the guest agent. The base image from which the guest machine is booted needs to be prepared before the automated analysis. It is basically just a fresh Windows 7 installation with all required drivers installed on it and all available updates applied to it. The only extra application that we installed on the base image was the guest agent, which the analysis system requires.

Cloning the base image every time a guest machine is launched is time consuming as the image may be gigabytes in size. Thus, the guest machines use differential file systems on top of an immutable base image. The use of differential file system brings several benefits. Firstly, a clean OS installation may be tens of gigabytes in size. Thus, the use of differential file system brings potentially significant performance improvements as it removes the need to clone the entire disk every time a guest machine is initialized (creating a new differential file system on the other hand the requires very few write operations). Also, it naturally reduces the need for storage space thus decreasing the cost of building the analysis system. Finally, if the analyzed applications do not perform many writes, it is possible to store the entire differential disk in the memory to speed up the analysis. A small differential disk can also be stored in its entirety for later analysis. This would not be practical without the use of differential file system when analyzing hundreds of application.

As already discussed, the guest machines may be virtualized or run on bare metal. When using the bare metal approach (depicted in Fig. 3c) the booting process is slightly more complicated than with virtual machines, which are easily configured to use differential file systems and easy to restart as



(a) Single machine (VirtualBox) (b) Cloud supported (OpenStack) (c) Bare metal

Fig. 3 – Different deployment strategies for the analysis system. Current implementation supports virtualized guest machines with VirtualBox (a) and OpenStack (b), as well bare metal guest machines (c).

well as power on and off. Instead of having physical hard disks on the bare-metal guest machines, the immutable base image is stored on the same physical host that also runs the host controller. The differential file system, which are attached to the bare-metal guest machines, are created on demand by the host controller when the guest machines are booted. The detailed boot sequence for a bare-metal guest machines is following. First, the guest machine is configured to boot from a USB stick, which contains a custom Preboot eXecution Environment (PXE). The script on the PXE creates a HTTP request to the host controller, which triggers the creation of a new differential VHDX disk on top of the immutable base image. After this, the host controller will expose the newly created disk to the guest machine over iSCSI and gives the guest machine instructions to boot from that image. Following this, the booting process continues as with virtual guest machines.

There are several benefits in using diskless bare-metal guest machines. As already discussed, using differential file system significantly increases the analysis system performance and reduces the need for storage space thus decreasing the deployment cost. Moreover, the diskless bare-metal guest machines are more resilient against hardware failures. That is, having no disks on them makes it is possible to hard-reboot the machines without hardware hazard. On the downside, relying on iSCSI requires fast network connection between the guest and host machines. Also, the host controller has to implement iSCSI target services as well as virtual disk management services. Currently, our implementation supports this only when the host controller runs on a Windows Server.

As far as we are aware, no other open-source bare-metal sandbox-system has similar feature. That is, the use of differential file systems on the guest machines differentiates our analysis system from other malware analysis sandboxes that support bare-metal analysis (e.g. Kirat et al., 2014).

The host controller is also responsible for the end side of the guest machines' life-cycle. When the host controller detects that an installer analysis has finished, it starts cleaning the guest machine. There are two different events that may trigger the cleaning. First, if the installer process and all subprocesses it has spawned have exited, the installation is deemed finished. Alternatively, after 20 min timeout, the guest agent starts sending the collected data to the host controller and attempts to gracefully shutdown the system. If this does not succeed, the host controller performs a hard shutdown after 30 min. On bare-metal guest machines, the hard shutdown was implemented with TP-Link HS100 smart plugs through which they draw their power. This also requires enabling a BIOS setting that restarts the guest machine once it regains power – this way the guest machine boots up once the smart plug is switched on. Because the bare-metal guest machines are diskless, the hard power-off does not risk the hard-drives.

Finally, when the results have been collected from the guest machine and it has been powered off, the guest machine can be initialized to analyze a new binary. If the guest machine runs as a virtual machine, this is done through the hypervisor's APIs. With bare-metal guest machines, the differential file systems are simply reinitialized.

3.4. *Installer execution and interaction*

The actual installer analysis can start after a guest machine has been set up and started. Right after the OS has booted, the guest agent requests an installer binary from the host controller. This happens over a HTTP connection. The downloaded installer is then executed. The installer, naturally, requires user attention. Therefore, we implemented a heuristic interaction system which emulates the behavior of a lazy user during the installation process. When the installer runs, the guest agent detects when it is waiting for user input and then sends the input event that is most likely to cause progress. It does this by trying identify and rank potential buttons shown on the UI. As an example, button-like shapes containing labels such as "Next", "Finish", and "Install" are preferred over labels such as "Cancel" or "Quit". The UI interaction heuristics in the guest agent were optimized for Windows; however, they could easily be adapted to other operating systems. We explain the UI-automation mechanisms in detail in Section 4.

3.5. *Installer monitoring*

We monitored the installer throughout the installation process. The system collects data at two different places. First, all system modifications, including file and registry writes, are logged within the guest machine. Second, all network traffic to and from the guest machine is logged in the external sniffer. We now discuss these in order.

Table 1 – Hooked functions in `ntdll` and `kernel32`.

Library/ service	Function	Use of the function
`ntdll`	NtCreateFile	Create file
`ntdll`	NtOpenFile	Retrieve handle to file
`ntdll`	NtDeleteFile	Delete file
`ntdll`	NtSetInformationFile	Alter file metadata
`ntdll`	NtCreateKey	Create key in the registry
`ntdll`	NtCreateKeyTransacted	Creates key in registry
`ntdll`	NtOpenKey	Get handle to registry key
`ntdll`	NtOpenKeyEx	Get handle to registry hive
`ntdll`	NTClose	Close file/registry handle
`kernel32`	CreateProcessInternalW	Spawn new process
`kernel32`	ExitProcess	Clean process resources

3.5.1. *Monitoring modifications to guest machine*

The level on which the system resources are monitored is a tradeoff. Monitoring at a lower layer, closer to the hardware, makes it more difficult for an application to evade the detection. An extreme approach would be to implement the monitoring by modifying the device drivers, or when the guest machine is virtual, in the hypervisor. That, however, would require intimate knowledge of the underlying hardware and associated communication protocols, and the lack of abstractions at the lower layer would reduce the quality of collected data and make data analysis more difficult. For a suitable balance, we placed our monitoring agent between the user and kernel-space.

We used API hooking by Microsoft Detours (Hunt and Brubacher, 1999) to monitor the API exposed by `ntdll`, which every Windows process is required to load (Faircloth et al., 2006). (If hooking to the `ntdll` did not succeed for some reason, the analysis system exits). Detours re-routes selected `ntdll` function calls to our logging functions. More specifically, the inline hooking in Detours overwrites the first instruction of library functions after they have been loaded into the process memory, so that our monitoring code is executed within the process before and after certain `ntdll` function calls. Thus, inline hooking can target selected processes, which are the installers in our case.

It is not sufficient to hook `ntdll` functions on the installer process that is being analyzed because installers often spawn new processes or use Windows services. Thus, we hooked on functions that spawn child processes and recursively applied hooking on them. Also, we hooked to two system services, Distributed COM (DCOM) and Microsoft Installer Service (MSI), which are often used by installers. Table 1 lists all the functions on the libraries and services which the current implementation hooks on. (Future releases of the analysis system will extend the hooking, for example, to Transaction Based APIs, which are used in Windows 7, 8 and 10).

Since the goal is to monitor changes to the target system, we only log data on operations with the write permission. That is, whenever a registry or file is either opened, created or deleted, we check whether the operation is granted write permission. If so, the hooked process is blocked while the analysis system logs information about the event and computes the hash of the object that will be modified. For example, upon a call to NtOpenFile, the system calculates the hash of the file before the access. The later write calls (e.g. with NtWriteFile) are not logged, thus avoiding excess data. Finally, when the file is closed with NtClose, we again calculate the file hash. That is, we keep track of file versions but not individual file writes. Operations on registry keys are logged similarly: we log the registry values when a registry key is opened and when it is closed.

The DLL hooking is the most OS-dependent part of the analysis system. We currently support 32-bit Windows 7, and are developing support for 64-bit Windows 7, which are well documented and has commonly been used for malware analysis. Nevertheless, it seems that only few adjustments would be needed to support newer Windows versions. Similar monitoring could well be implemented for Linux guest machines, but the mechanisms and collected data formats would be quite different.

3.5.2. *Network sniffing and traffic analysis*

We implemented network sniffing on an external virtual machine, which also provides basic network services (DHCP, DNS, and NAT) for the guest machines. When using bare-metal guest-machines, the sniffer is located on a dedicated physical host. The sniffer logs all DNS requests and network connections from the guest machines. To gain access to encrypted streams, we put our own root certificate on the guest machines and HTTPS proxy on the sniffer to capture all connections to port 443. Unlike other parts of the analysis system, the sniffer was implemented on Linux, taking advantage of its built-in networking services.

The sniffer logs all traffic with *tcpdump* into a *pcap* file. During traffic post-processing, we use *tshark* to extract L3 information such as connections and peer hosts. For HTTP connections, we use *tcpflow* to reconstruct response bodies and downloaded files. We then calculate their hashes. We also recursively calculate the hashes of files in compressed downloads. This provides information about the exact source of files installed on the guest machine.

3.6. *Analysis system performance*

To test the performance of the system, we used it to analyze 1177 installers (we explain how the installers were chosen in Section 5). Automatic execution of the analysis on the entire input set required 36 h, using 8 virtual machines on a single HP Proliant BL280c G6 blade server. The blade was equipped with twin quad-core Intel Xeon e5640 processors and 64 DDR3 GB RAM. To test scalability, the analysis was repeated with two identical blade servers, one running eight virtual machines on Windows Server 2012 x64 and the other six virtual machines on Ubuntu Server 10.14 x64. The latter test required 21 h to complete, demonstrating both scalability and multiplatform compatibility. This translates to 1 installer/min throughput. More recent hardware (e.g. high-speed solid-state drives) could further improve the performance of the system.

We also analyzed the same set of installers with a bare-metal setup. More specifically, we used four DELL Optiplex 960, each one running a single Intel Q9550 processor and having 8 GB of DDR3 memory. The machines did not have physical hard drives. Instead, we manually configured them to boot

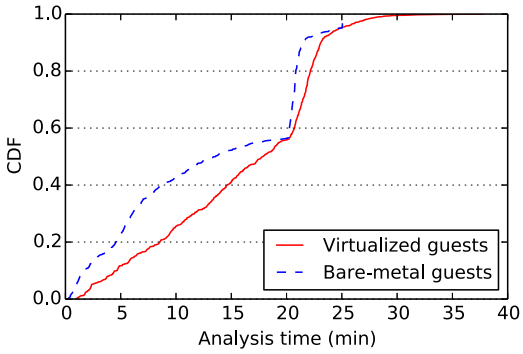


Fig. 4 – CDF showing the analysis time for a single installer.

from USB flash memory sticks. In order to implement hard rebooting of the guests, the machines drew power through TP-Link HS100 smart plugs. A fifth DELL Optiplex 960 (with same configuration, but with physical hard drive installed) was used as hardware sniffer. Host controller and database were hosted on a Fujitsu ESPRIMO D556 with 32 GB of DDR3 memory and 500 GB SSD.

Fig. 4 shows the CDFs of the analysis times with virtualized and bare-metal systems. With the virtualized system, the average analysis time for an installer was 15 min 15 s and median 17 min 53 s. With bare-metal setup, the times were 13 min 9 s and 13 min 36 s, respectively. With both system setups, the CDF shows a steady raise between 0–20 min after which it bounces sharply. This is caused by the 20 min timeout, after which the system attempts to retrieve all collected data and tries to gracefully shut down the guest machine unless the guest agent still tries to interact with the installer.

The relatively high average analysis time is caused by the way we monitor the completion of the installation. That is, we monitor every process spawned by the installer and deem an installation finished only when all of the spawned processes have exited. However, often the installers either start the application or open up some webpage at the end of the installation. In the worst case, this causes the analysis not to finish before the 30 min hard timeout occurs and the guest machine is powered off. Because of this, the analysis system should be scaled horizontally rather than vertically, that is, by increasing the number of guest machines instead of improving the performance of the individual hardware parts. This can also be seen by comparing our experiments done with virtualized and bare-metal systems: while the bare-metal environment does outperform the less powerful virtualized analysis system, in both cases about 40% of the installers do not finish before the timeouts.

4. Automated UI interaction

This section explains how we automated the interaction with the installer UIs. The topic deserves its own section because there are many GUI frameworks and we try to deal with all of them at once.

4.1. *GUI frameworks and automation*

There are several GUI APIs for Windows, such as the *Winforms* library in the .NET framework (Microsoft, c). There are also multiplatform UI libraries, such as Sciter and *Qt* (Blanchette and Summerfield, 2008). In Sciter, which is increasingly used by installers, the front-end is implemented with HTML5 and JavaScript. These libraries differ greatly from each other both in their look-and-feel and how they function under the hood.

There are also various libraries for automating UI interactions, such as the Microsoft Spy++ (Brenner, 2007) and Snoop[5]. Some GUI frameworks, like Sciter, provide their own inspection tools, which are mainly meant for debugging.

There are two problems with these inspections tools. Firstly, the amount of data they generate can be overwhelming. Secondly, they are specific to a particular UI framework and cannot detect the custom controls or messaging of other frameworks. Sciter is among the most difficult frameworks to automate because it only registers one monolithic window to the OS and handles all messages with an internal event handler Sciter.

To summarize, no single inspection framework can provide exhaustive information when dealing with advanced graphical interfaces. We therefore developed a more generic heuristic process for automating GUI interaction, which uses image processing, i.e. shape and text recognition, for identifying UI elements in a specific window. This enables a framework-agnostic interpretation of the GUI.

4.2. *UI interaction engine*

Our UI interaction engine resides inside the guest machine where the installer is run. It automates the software installation by emulating the behavior of a habituated user. That is, it follows the path of most obvious progress in the installation process. In the rest of this section, we explain the details of the process.

4.2.1. *Finding the active window*

The very first step performed by the engine is to check whether the installer process under analysis is still alive. If it is not, the engine terminates. Otherwise, it waits short moment to allow the UI to update and then identifies all the windows that are owned by the analyzed process. From these, the engine chooses the active window and analyzes its content, as described below.

4.2.2. *Detecting when to interact*

We first need to determine when a window is waiting for user input. There is no standard technique for this. The heuristic approach we found good is to observe when the UI becomes stable. Installers tend to show a progress indicator to reassure the user that work going on, but when the UI requires user input, the indicator stops changing. We exploit this observation as follows: when the active window does not change visibly for two seconds, it is likely to be waiting for user action.

[5] https://snoopwpf.codeplex.com/documentation.

The interaction engine implements heuristic by taking periodic screenshots of the active windows, applying edge detection algorithm on the image, and calculating a hash of the processed image. The image processing is necessary to ignore subtle changes caused, for example, by UI animations.

The UI stability detection can cause a false positive when a single-threaded installer become stuck due to heavy processing or blocked IO access. We avoid such mistakes by checking whether the installer process is active, i.e. whether it called the hooked `ntdll` functions. The ability to combine the visual and low-level API information was one of the reasons for developing our own guest agent instead of using existing sandboxes.

The window stability detection is used for three purposes. As already mentioned, it indicates when the UI is waiting for user input. Additionally, the hashes indicate whether an interaction with the UI was successful: if the previous interaction with the UI did not change it visibly, it is considered unsuccessful. This may happen, for example, if the engine tries to interact with an element that was misidentified as a button. A different interaction can then be tried. Finally, it is possible that the interaction with the UI goes to a loop. For example, a *Back* button can cause the automation to loop back to the previous screen. These loops are also detected by comparing the hashes of the screenshots. If the engine cannot get out of the loop, it terminates.

4.2.3. *Detecting UI elements*

Once the engine deems the UI stable, it tries to detect UI elements in the active window. As mentioned above, this is not trivial because the different frameworks handle low-level UI messages in different ways. Thus, we opted for a hybrid approach. On one hand, we use the Microsoft UIAutomation library to list native Windows UI elements (i.e. Winforms). On the other hand, we run our own visual recognition process to discover the UI elements of other frameworks.

The visual recognition is illustrated in Fig. 5b. First, the engine performs image preprocessing to remove color and to reduce the amount of detail. It then applies shape recognition (implemented with AForge.NET). The engine particularly tries to identify quadrilateral shapes that may be buttons. We then use OCR to extract textual information from the detected button-like objects (implemented with Tesseract OCR). Finally, the button positions and text data are combined with that from UIAutomation. In case of approximate duplicates, we prioritize the UIAutomation data because it also includes the type of the UI element and the supported interaction patters (e.g. whether it is a button or checkbox).

4.2.4. *Selecting UI element*

Once the interaction engine has detected the UI elements, it assigns scores to them based on their position and text. Native UI elements also have other properties such as whether they are disabled. These scores are summed up to get the final score for each element. The screenshot on Fig. 5c illustrates this with an example where the install button has received a score 290. This value comes because the common string "accept and install" is whitelisted giving the button a value of 280. The additional 10 points come from the location of the button: we found that the proceed buttons are usually located on the

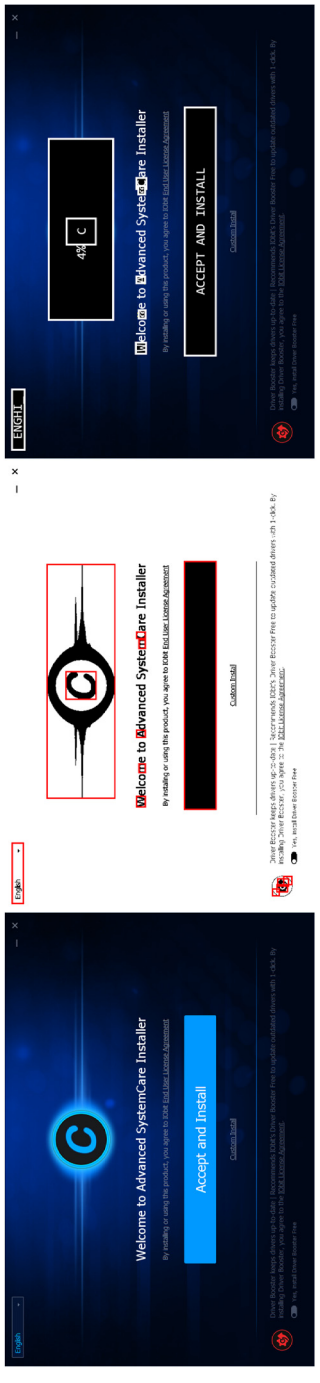


Fig. 5 – Visual UI element detection: (a) Original screenshot; (b) filtered screenshot, from which the hash is calculated, with shape recognition for potential buttons (c) buttons identified by combining shape and OCR.

Table 2 – Scoring rules for UI elements as well as the black and whitelisted words. The last four rules are only used with native Windows UI elements as this information is not available when using custom UI.

Property	Evaluation	Score
Text on element	null string	-30
	exact match to whitelisted string	280
	partial match to whitelisted string	30
	exact match to blacklisted string	-280
	partial match to blacklisted string	-30
Position	bottom right corner	10
	top left corner	0
	other scores interpolated	
Type of element	information not available	0
	button	50
	checkbox	15
	radiobutton	15
	hyperlink	10
Enabled?	element is enabled	0
	element is disabled	-1000
Checked?	checkbox/radiobutton is checked	-100
	checkbox/radiobutton is unchecked	50
Focused?	element is focused	50
	element is unfocused	0
Whitelist		
next, continue, [i] agree, [i] accept, ok, install, finish, run, done, yes, accept and install, next >		
Blacklist		
back, disagree, cancel, abort, exit, back <, decline, quit, minimize, no, close, pause, x, _, do not accept, <, [forward | backward | back] by [small | large] [amount]		

bottom right of the window. The scoring rules (see Table 2) were obtained experimentally through trial and error.

The interaction engine then chooses the element with the highest positive score and interacts with it. If no items are available or they all have a negative score (e.g. cancel button), the engine repeats the inspection every three seconds until a timeout or until all the monitored processes exit.

4.2.5. *UI interaction*

How the engine interacts with the chosen UI element depends on whether the element is a native Windows UI element or not. For native elements, the engine uses UIAutomation and an interaction that matches the type of the element. Our current implementation supports checkboxes, radio buttons and regular buttons. On the other hand, if the chosen UI element is not native, the engine simply simulates a mouse click event over the element.

After interaction, the engine waits for a reaction from the UI. This mimics the user behavior: the UI is expected to react to a successful interaction. If no reaction is observed within 500 ms, the engine assumes that the interaction failed and tries to interact with the UI element that received the next highest score, and so on until the interaction is successful or no elements with positive score remain. In both cases, the engine then starts whole process described in this section again from the stability detection.

Table 3 – Download portals studied in this paper.

Download portal	Alexa rank Oct.2016	Filters	Downloaded files	Successfully analyzed
download.cnet.com	159	Win,free	200	146
softonic.com	285	Win7,free	170	126
filehippo.com	662	Win	90	64
informer.com	881	Win,free	200	117
softpedia.com	1732	Win,free	200	148
majorgeeks.com	6077	Win,free	55	37
soft32.com	7279	Win,free	200	113
brothersoft.com	8600	Win,free	41	26
manual download	–	–	20	15
			1177	792

5. Large scale analysis of freeware installers

So far, we have described our analysis system as well as some design and implementation decisions we made during its development. We now report how the system was used to study freeware installers obtained from various download portals.

5.1. *Installer crawling*

We chose eight download portals based on their Alexa rankings (Table 3). While some of these sites also provide other content than application downloads, the ranking gives rough picture of their popularity and perceived trustworthiness.

Each studied download portal promotes a list of most popular applications on its front page, except Softpedia which promotes recent downloads. We decided to focus on the promoted applications and set a crawler to download up top 200 installers from each portal. When possible, it applied a filter for 32-bit Windows freeware. With some portals, there were fewer than 200 actual downloads, mainly because of the limitations of the web interface. Table 3 summarizes portals chosen for our study and the number of downloaded files.

In addition to crawling, we also manually downloaded installers for the most popular freeware applications directly from the developers' websites. We used Alexa rankings of top freeware applications as well as Google Trends for the most popular searches that include the words "software download". The manual download was done to compare the behavior of the installers from shareware authors with those provided by the download portals. However, it should be emphasized that we only downloaded 20 installers manually.

5.2. *Automated installer analysis*

As discussed in Section 3, the analysis system supports both bare metal and virtualized guest machines. We conducted this study with with guest machines running on VirtualBox, which ran on a Windows Server 2012 host machine in a local data center. Table 4 lists the software that the system used at the time of analysis.

Our UI automation engine was not able to automatically finish every installer properly. First, 10% of the crawled files failed either because the application was not an installer in

Table 4 – Used software platforms and libraries.

Node	Software component	Version
Host controller 1	Operating system	Windows 2012 R2 std 6.3.9600 (x64)
Host controller 2	Operating system	Ubuntu server 14.04 (x64)
Host controller 1 and 2	Crawling	Python 2.7 (x32) scripts
	Database	PostgreSQL 9.5 (x64)
	Logic	Python 2.7 (x32)
	Hypervisor	VirtualBox 5.0.18 (x64) + SDK
Sniffer	Operating system	Ubuntu server 14.04 (x64)
	Packet capture	Tcpdump 4.5.1
	MitM proxy	MitM proxy 0.17
Guest machine	Operating system	Windows 7 pro SP1 6.1.7601 (x32)
	API hooking	Microsoft Detours Express 3.0 (x32)
Guest agent	Runtime	.NET Framework 4.0
	Image processing	Aforge 2.2.5
	OCR	Tesseract 3.0.2

the first place (e.g. a stand-alone application) or because of missing hardware, software dependency, product key, or a similar reason. Additionally, 23% of the installers failed because the automated UI interaction was not smart enough. The reason was mostly complex interaction, such as selecting directory to which the program should be installed. Another reason was that the installer used some other language than English. The high percentage of successful installers (67%) is the result of iterative improvements to the UI automation and other parts of the analysis system.

The results discussed in the rest of this paper were obtained from the 792 installers completed successfully. Of these files, 751 were unique. We nevertheless consider even the installers with the same hash as distinct because some download portals have in the past served identical installers for several applications[6]. In these cases, the installer executable determined the further files to download and install based on its own filename.

6. Results

We present the results of our analysis in two parts. Section 6.1 describes what we can learn simply by looking at the files served by the download portals. Then, Section 6.2 presents the results of dynamic analysis. All the results are based on the 792 installers that were successfully executed. Some of the results are not directly related to security but are of general interest and serve as background information.

While we have published some of these results earlier (Geniola et al., 2017), the discussion presented here is more comprehensive than what has been published earlier.

6.1. *Static properties of the installers*

This section describes some of the basic properties of the analyzed installers.

6.1.1. *Analyzed applications*

We first compare the applications promoted on different portals. This helps to understand the data and is interesting in itself. We manually grouped the different versions of the same applications. Table 5 shows the overlap in applications at different portals. The number of distinct applications served by each portal is on the diagonal.

Our first observation is that the portals serve quite different sets of applications. Those promoted by CNET, FileHippo, Informer and Soft32 overlap the most. On the other hand, Softonic and Softpedia tend to promote applications that are not on the other portals. In the case of Softpedia, the reason may be that it does not promote the most popular software but the latest downloads. Finally, some portals use only the last week's downloads for the popularity ranking. This metric is susceptible to manipulation and short-term fluctuation, e.g. when an update is published. For these reasons, one has to be very careful when comparing different download portals based on our data.

6.1.2. *Application signing*

Our first security-related question was whether the applications are signed. Recent research showed that while malware is generally not signed, potentially unwanted programs are (Kotzias et al., 2015). We wanted to know where software distributed by the download portals stands.

Microsoft *Authenticode* is a digital signature system based on the PKCS#7 standard for signing executable file formats such as PE (Microsoft, a). Before executing a downloaded file, such as installer, the operating system checks whether it is signed by a trusted publisher. Upon the execution of the installer, Windows User Account Control (UAC) shows the user a different warning dialog depending on the signature. The dialog advices greater caution with unsigned installers as well as if the signature is not trusted or is invalid. However, it should be noted that the signature itself does not mean that the software publisher is trustworthy: signed code might behave maliciously and vice versa. It is up to the user to decide whether she trusts the publisher.

We analyzed the downloaded executables with the *Sigcheck*[7] utility. The results are shown in Table 6.

While most of the analyzed binaries (64%) had a valid signature, 30 (3.8%) cases did not verify correctly. Publisher certificate expiration was the most common cause of failure (24 cases). Other causes were explicit revocation (1 case) and untrusted root certificate-authority (5 cases). The remaining 32% of the analyzed installers were unsigned.

[6] CNET's downloader VT report available at https://virustotal.com/it/file/9961ebc9782037f68b73096bcff3047489039d6dc5c089f789b3dbff4109e21b/analysis/.

[7] https://technet.microsoft.com/en-us/sysinternals/bb897441.aspx.

Table 5 – Number of common applications served by each download portal pair. Different versions of same application have been combined.

	Brothersoft	CNET	FileHippo	Informer	MajorGeeks	Soft32	Softonic	Softpedia	manual
Brothersoft	26	1	3	2	0	0	1	0	0
CNET	1	144	19	22	6	21	7	0	4
FileHippo	3	19	64	18	6	15	4	1	4
Informer	2	22	18	117	7	14	3	0	5
MajorGeeks	0	6	6	7	35	3	1	0	1
Soft32	0	21	15	14	3	112	6	2	2
Softonic	1	7	4	3	1	6	125	2	0
Softpedia	0	0	1	0	0	2	2	148	0
manual	0	4	4	5	1	2	0	0	15
# distinct files	*26*	*146*	*64*	*117*	*37*	*112*	*126*	*148*	*15*

Table 6 – Signature verification of analyzed installers.

Verification outcome	# .EXE	# .MSI	# Total
Signed and verified	486	23	509
Verification Error	26	4	30
Unsigned	239	14	253

Interestingly, there were differences between the download portals. CNET, FileHippo and informer had about 80% correctly signed code while Soft32, Softonic and Softpedia had lower rates (62%, 61%, 44%, respectively). The rest appeared to belong to the latter group, but there were too few installers for a fair comparison. The high percentage of signed files in three of the four most popular download sites seems to indicate that there is value for the publishers in code signing even though the portals do not require it.

6.1.3. *Installer ages*

Software requires maintenance over the time. The new versions may add missing features, but they also patch discovered security vulnerabilities and other bugs. Thus, software age may be an indication of how seriously the publisher or portal take security. To map the situation, we investigated the age of the software in the crawled portals.

Given a binary file, there are a few ways of detecting its age. One possibility is to simply look at its metadata, relying on the last indicated modification date. Binary files also contain the timestamp of the linking process, added directly by the compiler, in the PE header. While that information may be accurate in most of the cases, there is at least a theoretical possibility of tampering. Instead of relying on information in the binary files themselves, we chose to use the *first-seen* date from VirusTotal[8]. Our assumption is that popular software tends to be submitted to VirusTotal soon after release. Although the first-seen date does not precisely tell how old a binary file is, it gives an indication of when the software began spreading more widely.

Table 7 depicts results of aging analysis for each portal. The overall observation is that much of the popular freeware is not frequently updated, and many installers are several years old. This can be a cause for concern.

Table 7 – Age in days of crawled binary files. Minimum age for each portal was zero days.

	$1^{st}q$	Median	$3^{rd}q$	Max
Brothersoft	38.25	953	1735.5	2978
CNET	28	111	428	3504
FileHippo	27.75	159.5	925.5	3397
Informer	187	604	1054	3329
MajorGeeks	6	8	279	3184
Soft32	81	574	1356	3528
Softonic	252.25	722.5	1487	2793
Softpedia	9	18	25	2806
manual	5.5	117	393	3328

The collected data shows that CNET, MajorGeeks and Softpedia serve relatively recent software installers while the rest of the portals serve considerably older binaries. In addition to the actual age of the software, the results could be explained by differences in which software the sites promote and the type of software that each portal distributes. For instance, there may be value to archiving popular legacy software that is no longer updated. But even considering such alternative explanations, we can still assert that the most popular download site CNET distributes relatively recent software. Its installer ages aligned closely to those of manually downloaded files, which can serve as a reference metric.

For more insight into the versions of applications distributed by different portals, we manually compared the version numbers reported in the UI (e.g. in About menu) by those applications that are served by multiple portals. As shown in Fig. 6, Informer and Softonic clearly fail in providing latest build versions while CNET, FileHippo, Majorgeeks and Soft32 perform generally better. The others have too few samples to judge.

6.2. *Dynamic analysis of installers*

This section prents results from the dynamic execution and monitoring of the installers.

[8] https://virustotal.com/en/about/.

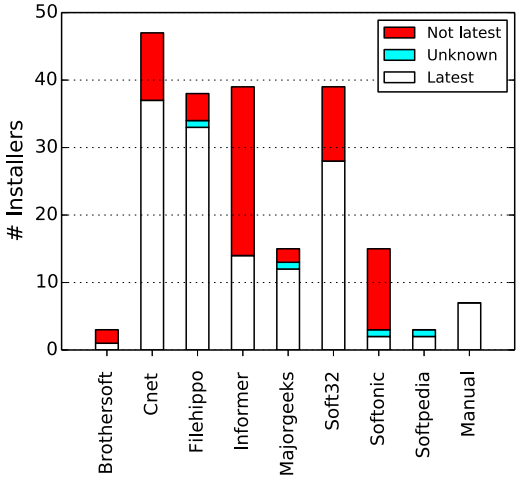


Fig. 6 – Version comparison between applications provided by multiple portals.

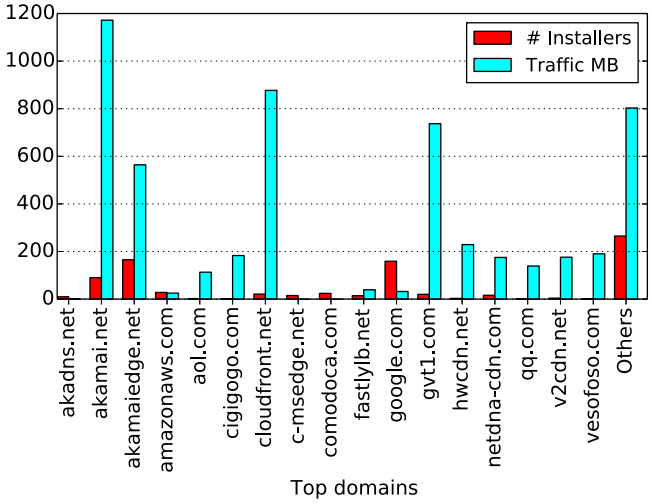


Fig. 7 – HTTP downstream traffic breakdown by top domains. The cyan bars (right) represent downstream traffic volume, while the red bars (left) indicate the number of installers that contacted the domain. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 8 – Breakdown of network traffic (inbound and outbound), total for all analyzed installers.

Transport layer	Application protocol	MB	(%)
TCP	HTTP	6567.84	95.22
	SSL	328.63	4.76
	others	1.25	0.02
		6897.72	99.25
UDP	UPnP	18.03	34.51
	SSDP	17.25	33.02
	DNS	4.39	8.40
	others	2.90	5.55
	unknown	9.67	18.52
		52.24	0.75
ICMP		0.01	0.00
Total		6949.96	100

6.2.1. *Network traffic analysis*

The sniffed traffic was analyzed with the Tshark and Bro protocol analyzers. We also implemented custom Python scripts for extracting further information.

We begin the discussion by looking at the network protocols (Table 8). Most of the traffic is HTTP and HTTPS over TCP (99%). The most frequent UDP packets were for UPnP, SSDP and DNS services. Our script was unable to classify some of the UDP packets. Manual investigation revealed that such traffic mainly belongs to the BitTorrent protocol, legitimately used by torrent-based installers. In three cases we identified JSON encoded text over UDP, used by content-sharing applications for advertising the system on the local network. The *360 Total Security* installer used a variant of the GVSP video streaming protocol. In the end, we were not able to classify 2.91 KB of UDP traffic generated by *PC Manager-Setup*, possibly due to custom binary encoding or encryption.

There are a few possible explanations for why HTTP is the preferred protocol. First, it usually enjoys best firewall traversal, even in restricted networks. Second, there are many stable and free HTTP client libraries available to developers, which provide one-line file download functionality and thus can speed up the development process of installers. Third, it is easy to move the HTTP server end to a content distribution network (CDN) for high availability and elasticity. This purpose is confirmed by the download site analysis, presented in Fig. 7; more than 80% of the HTTP downstream traffic is downloaded from well-known CDNs. Akamai and Google are the two most-contacted ones.

The high number of experiments contacting Google could indicate that many installer authors benefit from its value-added services such as user tracking and demographic data. In fact, 23 installers made least one HTTP request for the Google Analytics web beacon. Moreover, 29 installers downloaded the Google Analytics JavaScript library. This typically happens as the last step of the installation process when the installer opens a readme page with the user's default web browser. In these cases, the shareware author gets Google Analytics data on the users.

We reassembled and inspected the HTTP streams for binary content. Table 9 shows the results. Executable files and binary payloads constitute most of the traffic. This indicates that many of the installers (348) behave as install-time downloaders. To see if there is a clear distinction, we plotted the installer binary size and the downloaded traffic volume in Fig. 8. We have visually classified the installers into three classes: downloaders, installers that call home but do now download significant amounts of data, and stand-alone installers. (It should be noted that even the installers that did themselves generate any network traffic are shown to send 14 bytes of traffic. This is because Windows 7 makes a single HTTP request at the startup, in order to test the Internet connectivity).

6.2.2. *Man-in-the-middle vulnerability*

As seen in Section 6.2, installers tend to download binary files over insecure HTTP connections. These files are either

Table 9 – HTTP downloads by MIME type.

MIME type	Downloaded data (MB)	# Installers
application/x-dosexec	1879.17	96
application/octet-stream	1808.99	227
application/vnd.ms-cab-compressed	475,02	25
application/gzip	462.82	11
application/zip	267.32	32
application/vnd.ms-office	257.15	8
application/x-7z-compressed	228.09	4
application/x-bzip2	29.51	4
text/plain	16.90	787
text/html	12.33	138
others	18.13	155
Total	5455,42	788 (Distinct)

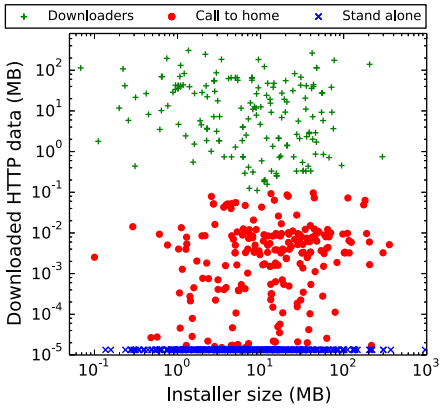


Fig. 8 – Scatter plot of installer size vs downstream traffic volume. The points were divided to the three classes based on visually observed grouping.

installed on the user's computer or executed within the installation process, possibly with high system privileges. In such a context, it is essential that the installers authenticate the downloaded files, e.g. with digital signature. To check if they do that, we implemented an automated MitM attack against the installers. This was done with a transparent HTTP proxy in the sniffer component of the analysis system, which replaced executable files in HTTP responses with its own. The malicious response was sent in the following cases:

1. Request URL ended with .EXE or .MSI
2. Response MIME type matched executable or MSI
3. First bytes of the HTTP response body matched magic numbers for EXE or MSI.

Upon successful MitM attack, a special EXE or MSI file is served to the client. When executed, that binary simply takes note of its running privileges and origin URL and terminates. Table 10 shows how installers treated the executable file served by the MitM attacker.

Among the 792 analyzed installers, the MitM attack was triggered 100 times. Amazingly, more than half of the attempted attacks (55%) led to immediate execution of the attacker's binary file, meaning that no authentication or integrity check was done for the downloaded binaries. Only 8 installers refused to execute the tampered file and removed it right away. In the remaining 37 cases, the attacker's file was not executed, yet it was found on the disk after the installation. 17 of these were saved in temporary system folders (subject to later removal upon system cleanup) while 20 were in persistent file system locations, such as under the *Program Files* directory. The latter cases leave the system open to a delayed attack when the application is used.

Table 10 – Outcome of the MitM attacks. Executed files are successful attacks. Saved files remained on the disk but were not executed automatically by the installer.

Download portal	Executed	Saved	Saved in temp	Removed	Total
Brothersoft	2	0	0	0	26
CNET	11	7	0	3	146
FileHippo	5	3	2	1	64
Informer	12	3	7	1	117
MajorGeeks	4	0	1	0	37
manual	1	3	1	0	15
Soft32	10	2	3	1	113
Softonic	9	0	3	1	126
Softpedia	1	2	0	1	148
Total	55	20	17	8	792

The MitM attack is especially dangerous because the attacker's file was always executed with the same privilege level of the installer application. In 75 out of the 100 successful attacks, the vulnerable installers require elevated privileges to work properly.

The MitM vulnerability occurs equally in all download portals. Softpedia seems to be less vulnerable because it servers more stand-alone installers that do not download executable files. Moreover, the highest attacker success ratio of 33% is for the manually downloaded binaries. This suggests that the MitM vulnerability is created by the original software authors rather than by the portals.

There are straightforward ways of mitigating the MitM vulnerability. One approach would be to use HTTPS for the download. Another possibility is to use asymmetric signature verified by the installer application with a static publisher public key. Clearly, there is no good technical excuse to be vulnerable.

It is worth noticing that most of the download portals distribute installers via HTTP in the first place, and the installer itself could be fake. The user, however, has the opportunity to check that the installer binary is signed by the correct publisher. In comparison, the MitM attack succeeds even if the user takes care to only execute legitimate, signed installer binaries from the correct publisher.

6.2.3. *File system analysis for malware drops*

As explained in Section 3.5, we collect the hashes of all files that are temporarily or permanently stored on the guest machine as well as hashes of files reconstructed from network traces. We looked up all the collected file hashes in VirusTotal, which aggregates results from various virus scanners. This

Table 11 – Threats ranked by VirusTotal detection rate. Some file names of the detected PUP files have been truncated.

Installer name or file name	PUP file name	PUP MD5 hash	#Positive scanner(s)	#Inst	Description	Source portal
Wondershare MobileGo for Android	`rootf.apk`	`b2ae43bd8058e31915e721154d91d306`	30	1	Android rootkit	Soft32
Free Video To Audio Converter 2016	`fusion.dll`	`731af19d29ebb9f89cbbd619d1b5d50a`	17	1	PUP InstallCore	CNET
`videora-ipod-600-setup.exe`	`videora.exe`	`45d66d0a292aecc2a5be955a19be699b`	15	1	PUP OpenCandy	Softonic
Mobogenie Installer	`...Setup.exe`	`2d18c828217d8d48ba8d6e0753b77936`	14	1	Adware Mobogenie	CNET
FreeYouTubeDownloader Setup	`fusion.dll`	`5d2f9778dad625ca4a63e363719101e6`	14	1	PUP InstallCore	CNET
Free Studio	`fusion.dll`	`c983e01e337523f46a80d9a25ed955f7`	12	1	PUP InstallCore	Soft32
Check Point Install Utility	`...0061e.exe`	`0f6b98dd6517c69ff60630b58f521f00`	12	1	PUP Montiera toolbar	Softonic
`bs_FormatFactory.exe`	`...B2C6B.dll`	`6fa248c7eff5519fb18d99d423c43978`	8	1	PUP Conduit	Brothersoft
PDFlite	`...126C1.exe`	`9c08b95823479b2b5a098a854d41c77a`	6	1	PUP Zugo Toolbar	Softonic

Table 12 – Most representative false positive detections from VirusTotal.

File name	Positive scanner(s)	#Inst	Description
*.zip	Tencent	76	Empty zip archive.
System.dll	Bkav	11	Legitimate DLL part of NSIS framework.
vlc-2.2.4-win32.exe	Baidu	3	Legitimate file belonging to Videolan player.
bass_mpc.dll	Bkav, Baidu	2	Legitimate DLL part of MusePack Audio Codec.

was done two months after running the installers to leave time for new malware variants to be detected.

The number of positive results was high (235 files) but most of these were reported by only one scanner and, most likely, were false positives. Only 1.3% of the installers contained files detected as malware by six or more scanners. More importantly, majority of such the positives were labeled as PUP. There was only one detected critical threat, and it was an Android rootkit that does not infect Win32 systems.

The files with highest detection rates are listed in Table 11. These include three versions of *fusion.dll*, which belongs to the *InstallCore*, an installer development kit that is considered to bundle unwanted programs.[9] Among the detected PUPs, *OpenCandy*, *Conduit*, *Mobogenie* and *Zugo* are known to be associated with PPI networks (Caballero et al., 2011).

The analysis shows that download portals are not used for blatant malware distribution. The portals probably perform scans of the binaries before publishing them. On the other hand, the presence of PUP related to a well-known PPI network supports the claims that download portals cloud have stricter countermeasures against grayware and bundled unwanted applications.

We include Table 12 because it may be of interest to others conducting similar experiments. It shows some typical false positives from the scanners. The most interesting case is an empty zip archive, which one scanner detected as malicious.

[9] Scan report available at https://www.sophos.com/en-us/threat-center/threat-analyses/adware-and-puas/Install&20Core/detailed-analysis.aspx.

Table 13 – Installers bundling unrelated third party browsers. Those marked as * mentioned third party software installation capabilities in EULA.

Product name	Publisher	Bundled browser
Adobe Shockwave Player*	Adobe Systems Inc.	Google Chrome
CCleaner	Piriform Ltd	Google Chrome
Defraggler	Piriform Ltd	Google Chrome
PhotoScape*	Mooii	Google Chrome
Recuva	Piriform Ltd	Google Chrome
Speccy	Piriform Ltd	Google Chrome
SUPERAntiSpyware Free	SUPERAntiSpyware	Google Chrome

6.2.4. *Registry modifications*

We tracked the registry modifications made by the installers and analyzed changes to the following:

- Automatic program startup
- Default browser
- Browser plugins

There are many ways for a program to start automatically in Windows including registry keys and specific file system folders (Microsoft, 2006). We found that 88 installers (12%) configured the installed software to automatically run at system startup.

Similar analysis was done on default browser changes: 26 installers replaced the default browser. Interestingly, 11 of these are not browser installers. Table 13 reports the details. Google Chrome turned to be the only third-party browser installed by non-browser installers. Manual investigation of

Table 14 – Third-party plugins dropped on IE.

Portal	#Installers	Toolbar	Menu extension	BHO	Total	%
Brothersoft	26	2	0	4	6	23.1
CNET	146	4	7	7	18	12.3
FileHippo	64	7	0	9	16	25.0
Informer	117	4	1	7	12	10.3
MajorGeeks	37	0	0	0	0	0.0
Soft32	113	2	0	4	6	5.3
Softonic	126	1	1	2	4	3.2
Softpedia	148	1	0	3	4	2.7
manual	15	1	0	2	3	20.0
Total	792	22	9	38	69	

the recorded screenshots revealed that, in all cases, Google Chrome installation is optional but pre-selected by default. We also looked at the EULAs proposed by such installers. At the time, EULAs for *Piriform* (Piriform, 2016) and *SuperAntiSpyware* (Support.com, 2014) failed to mention bundled software; neither is there an obvious technical reason that justifies browser installation. Some other installers did mention the presence of third party software offers. This suggests that there is a monetization scheme between Google and the software vendors which bundle Google Chrome (or earlier Google Toolbar) in their applications. Such scheme shares common characteristics with the PPI business model.

Our analysis continued with the inspection of installed third party browser modules. We focused on Internet Explorer (IE), which was the only browser installed by default on the fresh Windows 7 guest machine. There were 69 registry modifications regarding browser extensions by 38 installers. As shown in Table 14, the predominant type of installed extension is the *Browser Helper Object* (BHO), which is the most powerful and potentially most dangerous IE component type because it runs in the same memory context as the browser and has access to the user's browsing data (Esposito, 1999).

In the case of browser extensions, there were differences between the portals. MajorGeeks installers did not bundle any browser plugins, and Softpedia registered a total of just 4 dropped items over 148 installers (2.7%). Softonic and Soft32 had also relatively low rates. CNET and Informer, on the other hand, dropped considerably more browser plugins, and FileHippo topped the league with 25% drop rate. Interestingly, even manual downloaded installers bundled browser plugins. This could indicate that the plugins are bundled by the original software vendors and not by the portals. Some portals may actually be selective against such bundling.

6.2.5. *Installer best-practices compliance*

Microsoft advises vendors to follow certain best practices for installers (Microsoft, d). Firstly, each installed application should provide a consistent uninstall feature. For this, the installer should populate two registry keys on the system, one with the program's human-readable name and the other with a path to the uninstaller binary. If one of these two values is missing, removal of the application becomes cumbersome. Of the analyzed installers, 82 failed to specify both the program name and uninstaller path. Another 5 only stored the product name without specifying an uninstaller binary.

Microsoft also requires installers to specify valid *ProductName* property in their metadata, which is usually placed within the resource section of the executable file. It is exposed to the user in a properties dialog (Microsoft, b). 174 of the analyzed installers failed to provide this information.

7. Discussion

We have presented an analysis system for application installers. The system is fully automated and emulates user input based on image processing and heuristic rules on the UI screenshots. It supports virtualized environments, OpenStack data centers, and analysis on bare-metal hosts. We have demonstrated the performance of the system by analyzing 792 freeware installers in less than 24 h on an eight-core server. The execution time can be arbitrarily reduced by adding more hosts or processor cores.

The most immediate limitations of the current implementation are still in the UI automation. The rules for detecting the success and failure of the installation could be more reliable. Inspection of installation screenshots confirmed that the UI engine was able to correctly automate 67% of the installers. However, most of the analysis failures were due to incorrect UI interactions. There clearly is still scope for improvement.

Another limitation in our system relates to detecting whether an installation has been finished. This problem is not trivial. We tackled this with a rather simple approach that prefers data quality over analysis speed. That is, the system will continue the analysis until all monitored processes (i.e. processes spawned by the installer) have exited, or a timeout is hit. Because of this, a number of installers did not finish before the timeout. This was caused by the fact that it is very common for an installer either to start the application, or to open some webpage, at the end of the installation. We are confident that future improvements on this matter might drastically reduce the analysis time.

Third main limitation in the system is that it cannot accurately detect whether installers' behavior is actually wanted or not. That is, while the system monitors the changes that an installer does on the system, it cannot deduce whether these changes are actually unwanted. It might be possible to use natural language processing in order to reason what kinds of changes the user might expect an installer to perform. This kind of approach has earlier been demonstrated with mobile applications (Pandita et al., 2013; Qu et al., 2014).

Like many free automated malware analysis tools, the current system currently only supports 32-bit Windows 7 on the guest machines. This is because of the free version of the API hooking library we initially chose to use only supports 32-bit Windows. We have already started to implement support for 64-bit guest machines.

In this paper, we have focused on grayware PUP and explicit changes to the system. It is possible for evasive malware to detect the API hooking and change its behavior accordingly in order to hide itself. Using bare-metal guest-machines may enable the analysis of more evasive malware at least to some extent. However, it should be noted that because we rely

on well known API hooking libraries, our system is not completely stealthy even when using bare-metal guest-machines.

This research was motivated, in part, by the suspicion that download portals might distribute malware or bundle excessive amounts of unwanted programs to freeware downloads. The analysis that we did on the 792 installers does *not* support these suspicious. We did not find serious malware infections, and the bundled PUPs seems to come mostly from the original freeware authors.

From the security viewpoint, the most important analysis results were:

- The median age of installers varies notably among portals, and the distribute freeware versions are often not the latest.
- Some portals host installers that bundle known PUP.
- The most common types of PUP bundled with freeware are third-party browser plugins and the Google Chrome browser.
- Many installers that download executable files are vulnerable to MitM attacks that enable code injection to the client machine.

While we make some comparisons between the portals throughout the paper, it would not be possible to make fair ranking of the portals regarding security or PUP. The portals differ in the types and quantity of software available. While Softpedia does well on all the metrics, it promotes a different set of software than the other portals (based on recent downloads rather than popularity), and thus the results may not be comparable.

8. Conclusion

The contributions of this paper are twofold. On one hand, we have designed and implemented an automated analysis system for software installers and applied it to a large number of freeware from download portals. On the other hand, we demonstrate it by analyzing 792 installers with it. This shows that the analysis system is capable for fully automated installer analysis, which includes emulating the user-interactions required to finish installation procedure. We have published the analysis system as well as its documentation as open source.

The results from the 792 analyzed installers indicate that download portals do not actively distribute known malware. Many freeware installers, however, come bundled with unwanted programs and have links to PPI networks. Our most dangerous finding related to the way how application installers download application binaries during the installation process. The integrity (or the authenticity) of the downloaded binaries are often not verified and, as a consequence, several of the analyzed installers are vulnerable to man-in-the-middle attacks that enable code injection to the client machines.

Acknowledgments

This work was supported by Tekes as part of the DIMECC Cyber Trust program and by Academy of Finland (Grant number 296693).

REFERENCES

Blanchette J, Summerfield M. C++ GUI programming with Qt4. Pearson Education; 2008.

Böhme R, Köpsell S. Trained to accept? A field experiment on consent dialogs. Proceedings of the SIGCHI conference on human factors in computing systems. New York, NY, USA: ACM; CHI '10; 2010. p. 2403–6. doi: 10.1145/1753326.1753689.

Boldt M, Carlsson B. Privacy-invasive software and preventive mechanisms. Proceedings of international conference on systems and networks communications, 2006. ICSNC '06; 2006. p. 21. doi: 10.1109/ICSNC.2006.62.

Boldt M, Jacobsson A, Lavesson N, Davidsson P. Automated spyware detection using end user license agreements. Proceedings of international conference on information security and assurance, 2008. ISA 2008.; 2008. p. 445–52. doi: 10.1109/ISA.2008.91.

Brenner P. Spy++ internals. Available at https://blogs.msdn.microsoft.com/vcblog/2007/01/16/spy-internals/; 2007. [Accessed on 19 July 2016].

Bruce J. Defining rules for acceptable adware. Proceedings of the fifteenth virus bulletin conference, 2005.

Caballero J, Grier C, Kreibich C, Paxson V. Measuring pay-per-install: the commoditization of malware distribution. Proceedings of the 20th USENIX conference on security. Berkeley, CA, USA: USENIX Association; SEC'11; 2011. p. 13.

Emm D, Unuchek R, Garnaeva M, Ivanov A, Makrushin D, Sinitsyn F. Technical Report. IT threat evolution in Q2 2016; 2016. Available at https://securelist.com/files/2016/08/Kaspersky_Q2_malware_report_ENG.pdf.

Esposito D. Browser helper objects: the browser the way you want it. https://msdn.microsoft.com/en-us/library/bb250436(v=vs.85).aspx; 1999. [Accessed on 29 December 2016]

Faircloth J, Beale J, Temmingh R, Meer H, van der Walt C, Moore H. Penetration tester's open source toolkit. Elsevier Science; 2006.

Geniola A, Antikainen M, Aura T. A large-scale analysis of download portals and freeware installers, in: Lipmaa H, Mitrokotsa A, Matulevicius R (Eds.), Secure IT Systems, NordSec. Lecture Notes in Computer Science, vol. 10674. Cham: Springer; 2017. doi:10.1007/978-3-319-70290-2_13.

Good N, Dhamija R, Grossklags J, Thaw D, Aronowitz S, Mulligan D, Konstan J. Stopping spyware at the gate: a user study of privacy, notice and spyware. Proceedings of the 2005 symposium on usable privacy and security. New York, NY, USA: ACM; SOUPS '05; 2005. p. 43–52. doi: 10.1145/1073001.1073006.

Goretsky A. Technical Report. Problematic, unloved and argumentative: what is a potentially unwanted application (PUA)?; 2011. [Accessed on 03 June 2016].

Guarnieri C, Tanasi A, Bremer J, Schloesser M. The Cuckoo sandbox. 2012. Available at: https://cuckoosandbox.org/

Heddings L. Stop testing software on your PC: use virtual machine snapshots instead. Available at: http://www.howtogeek.com/206286/stop-testing-software-on-your-pc-use-virtual-machine-snapshots-instead/; 2014.

Heddings L. Technical Report. Yes, every freeware download site is serving Crapware (Here's the proof); 2015. Available at http://www.howtogeek.com/207692/yes-every-freeware-download-site-is-serving-crapware-heres-the-proof/.

Hunt G, Brubacher D. Detours: binary interception of win32 functions. Proceedings of the 3rd conference on USENIX Windows NT symposium - volume 3. Berkeley, CA, USA: USENIX Association; WINSYM'99; 1999. p. 14. url: http://dl.acm.org/citation.cfm?id=1268427.1268441.

Kirat D, Vigna G, Kruegel C. Barecloud: bare-metal analysis-based evasive malware detection. Proceedings of the 23rd USENIX security symposium (USENIX Security 14); 2014. p. 287–301.

Kotzias P, Bilge L, Caballero J. Measuring PUP prevalence and PUP distribution through Pay-Per-Install services. Proceedings of the USENIX security symposium, 2016.

Kotzias P, Matic S, Rivera R, Caballero J. Certified PUP: abuse in authenticode code signing. Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM; 2015. p. 465–78.

Lengyel TK, Maresca S, Payne BD, Webster GD, Vogl S, Kiayias A. Scalability, fidelity and stealth in the Drakvuf dynamic malware analysis system. Proceedings of the 30th annual computer security applications conference. ACM; 2014. p. 386–95.

McFedries P. Technically speaking: the spyware nightmare. IEEE Spectr 2005;42(8):72. doi: 10.1109/MSPEC.2005.1491233.

Microsoft. Introduction to code signing. https://msdn.microsoft.com/en-us/library/ms537361(v=vs.85).aspx; a. [Accessed on 31 December 2017].

Microsoft. VERSIONINFO resource. https://msdn.microsoft.com/en-us/library/aa381058.aspx; b. [Accessed on 05 January 2017].

Microsoft. Windows forms. Available at https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx; c. [Accessed on 23 July 2016].

Microsoft. Windows installer and logo requirements. Available at https://msdn.microsoft.com/en-us/library/windows/desktop/aa372825(v=vs.85).aspx; d. [Accessed on 30 December 2016].

Microsoft. Run, RunOnce, RunServices, RunServicesOnce and Startup. Available at https://support.microsoft.com/en-us/help/179365/info-run-runonce-runservices-runservicesonce-and-startup; 2006. [Accessed on 08 December 2016].

Motiee S, Hawkey K, Beznosov K. Do windows users follow the principle of least privilege? investigating user account control practices. Proceedings of the sixth symposium on usable privacy and security. ACM; 2010. p. 1.

Pandita R, Xiao X, Yang W, Enck W, Xie T. Whyper: towards automating risk assessment of mobile applications. Proceedings of USENIX security symposium, 2013.

Piriform. Software license agreement for users of CCleaner free software. Available at http://www.piriform.com/legal/software-license/ccleaner; 2016. [Accessed on 28 December 2016].

Qu Z, Rastogi V, Zhang X, Chen Y, Zhu T, Chen Z. Autocog: measuring the description-to-permission fidelity in android applications. Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. New York, NY, USA: ACM; CCS '14; 2014. p. 1354–65. url: http://doi.acm.org/10.1145/2660267.2660287, doi: 10.1145/2660267.2660287.

Sciter. Embedding principles. Available at http://sciter.com/developers/embedding-principles/. [Accessed on 19 July 2016].

Slade. Technical Report. Mind the PUP: top download portals to avoid; 2015. Available at http://blog.emsisoft.com/2015/03/11/mind-the-pup-top-download-portals-to-avoid/.

Statcounter. Desktop operating system market share worldwide. Available at: http://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-201706-201706-bar; 2017.

Support.com I. Superantispyware product license terms. Available at https://www.superantispyware.com/eula.html; 2014. [Accessed on 28 December 2016].

Thomas K, Crespo JAE, Rasti R, Picod JM, Phillips C, Decoste MA, Sharp C, Tirelo F, Tofigh A, Courteau MA, Ballard L, Shield R, Jagpal N, Rajab MA, Mavrommatis P, Provos N, Bursztein E, McCoy D. Investigating commercial pay-per-install and the distribution of unwanted software. Proceedings of the 25th USENIX security symposium (USENIX security 16). Austin, TX: USENIX Association; 2016. p. 721–39. url: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/thomas.

Willems C, Holz T, Freiling F. Cwsandbox: towards automated dynamic binary analysis. IEEE Secur Priv 2007;5(2):32–9.

Wood P, Nahorney B, Chandrasekar K, Wallace S, Haley K, et al. Technical Report. Symantec internet security threat report trends for 2016. Symantec Corporation; 2016.

Alberto Geniola received the B.Sc. degree in computer engineering from University of Bologna in 2013. He spent 4 months as an intern in the Security Lab, at UCSB, United States. He received the M.Sc. degree in computer engineering from Polytechnic of Turin, Turin, Italy, in 2017. During his master studies, he made a research visit to Aalto University in 2017, where he wrote his thesis about unattended analysis of freeware binaries. He currently works as a cloud engineer at Revevol SARL, Milan (Italy).

Markku Antikainen received the M.Sc. degrees in security and mobile computing from Aalto University, Espoo, Finland, and the Royal Institute of Technology, Stockholm, Sweden, in 2011. In 2017, he received a Ph.D. degree from Aalto University, Espoo, Finland. His doctoral thesis was on the security of Internet-of-things and software-defined networking. He currently works as a post-doctoral researcher at Helsinki Institute for Information Technology, Finland.

Tuomas Aura received the M.Sc. and Ph.D. degrees from Helsinki University of Technology, Espoo, Finland, in 1996 and 2000, respectively. His doctoral thesis was on authorization and availability in distributed systems. He is a Professor of computer science and engineering with Aalto University, Espoo, Finland. Before joining Aalto University, he worked with Microsoft Research, Cambridge, U.K. He is interested in network and computer security and the security analysis of new technologies.