Preventing Ransomware Attacks Through File System Filter Drivers

Giovanni Bottazzi, Giuseppe F. Italiano, Domenico Spera
University of Rome "Tor Vergata"
gbottazzi73@gmail.com, giuseppe.italiano@uniroma2.it, speradomenico@gmail.com

Abstract

Over the last years ransomware attacks have been widely spreading over the Internet, indiscriminately targeting home users as well as corporates and public agencies. Several approaches have been proposed to analyze and detect ransomware intrusions in literature, moving from combined heuristics, behavior analysis, sandbox-based solutions and machine learning techniques to function calls monitoring. Our approach differs from the above by shifting the focus from removing the problem to mitigating damages, to ensure data availability despite malware attacks. The aim is not to detect new ransomware samples, but simply to protect integrity and availability of private data. In other words, we interfere with ransomware usual behavior, intercepting I/O request packets and denying operations on user's valuable data.

1 Introduction

After 30 years from the first release, Windows systems continue to maintain desktop market dominance with more than 80% of worldwide users as shown in Table 1. Such spreading makes Windows operating system the main target of cybercriminals. According to Symantec (Symantec, ISR, 2016), only in 2015, more than 430 million of unique malware samples have been discovered, 36% up from the year before and specially a new form of cyber extortion is growing: ransomware detection has recorded 100 new families. Ransomware thread is the final evolution of criminal weapons (Savage et al., Symantec WP, 2015), using modern encryption algorithms and untraceable payment systems (Nakamoto, bitcoin.org, 2008), enciphering victim's files and demanding a fee to restore usability. At the end of 2015, the average ransom demand has raised to 679$. A recent research (Osterman Research, 2016) reports that nearly 40% of all organizations surveyed, experienced a ransomware attack in the same year. Even worst more than 600 million of attacks have been recorded in 2016 (SonicWall Report, 2017), well over 100 times compared to 2015.

Despite these big numbers, home uses still rely to basic computer security solutions, completely ignoring operating system security as well as third party vulnerabilities. Table 2 shows how they are

CEUR-WS.org/Vol-2058/paper-08.pdf

not inclined in changing their habits: Windows XP is still used despite known security concerns and updated versions like Windows 8 and Windows 8.1 have been avoided, probably due to a GUI “new look”.

Source	Windows	OS X	Linux	Others
statcounter	84.34%	11.68%	1.54%	2.44%
netmarketshare	91.59%	6.27%	2.14%	0.00%

Table 1. Desktop Operating System Market Share Worldwide. This statistic shows the operating system market share worldwide on Desktop PC during March 2017. It is based on aggregate data collected by statcounter.com and netmarketshare.com.

Source	Win 10	Win 8.1	Win 8	Win 7	Win XP	Others
statcounter	34.25%	9.62%	2.44%	47.06%	5.47%	1.16%
netmarketshare	26.53%	6.96%	5.83%	51.70%	7.78%	1.20%

Table 2. Desktop Windows Versions Market Share Worldwide.

More than that, corporates as well as public agencies focus on security technical aspects, leaving out human aspects (Furnell et al, Computer & Security, 2012): while technology is still a necessary investment, however, training and education must be part of the solution. Too often employees have been the entry point for a successful attack (Choi et al., IJFP, 2016; Forand, NJ.com, 2015; Green, Becker’s Hospital Review, 2016). On the other side the ransomware spread, first appeared in 1980 with the AIDS Trojan, has dramatically increased over the last years. They moved from the first families of *Legacy Crypto Ransomware* in 2005 (Gazet, JCV, 2010), lacking on accurate and strong encryption system implementation, through *Fake Antivirus* [Symantec, Report, 2009; Stone-Gross et al., Economics of Information Security and Privacy III, 2013), where payment is required to buy a license and solve nonexistent issues, to *Modern Crypto Ransomware* (Symantec, Report, 2016; Dingledine et al., USENIX, 2004), whose business model has been incredibly refined in many ways: no fake warning messages or false claims, updated and combined crypto algorithms, anonymous communications even through TOR, payments made through crypto currency, etc. The purpose of the present paper, aims to strongly mitigate ransomware attacks on Windows systems creating a mapping between mostly used private file types and referred default programs, not allowing other executables to modify such data. The focus on data protection from unwanted interactions, not taking into account the evolution of ransomware, may result non-innovative in a first analysis. However, the key concept on which we based the work is closely related to the high depth of the protection software layer developed within the Windows File System I/O stack. In other words the software layer implemented is positioned at a lower level of abstraction than the antivirus, thus resulting highly effective and difficult to bypass. Ideally, our solution could be used in enterprise environments, restricting access to private data only to known software as well as building a *totally locked down Windows System* for POS, ATM and Kiosk Mode Environments.

2 Related work

Malware Analysis is the action of proving that a suspicious program represents a threat to users. Numerous techniques exist to precisely analyze malware and deeply understand its behavior (Egele et al., ACM Computer Surveys, 2012), a necessary task to update anti-virus software with a new signature. New threads and variants over existing families give a trace of malware evolution: it’s a chess game where analysts play a defense strategy in order to give quick answers and to avoid wide propagation. In addiction malware authors use evasion techniques to avoid detection. Such a

condition, leads to a constant struggle between adversaries by using custom and joint evasion techniques from both sides. Obviously, ransomware can be considered as malware to all intents and purposes. In this context two main streams can be taken into account: static and dynamic malware analysis. Static analysis consists in software inspection without executing. When source code is available, this technique permits a complete understanding of given software. In a real scenario, analysts expect only a binary representation in order to extract useful information: a reverse engineering operation is necessary to obtain the assembly code. Furthermore, the malware authors have designed different obfuscation techniques especially in countering static analysis (Moser et al., ACSAC, 2007). Dynamic analysis, instead, consists in behavioral monitoring of suspicious software while it is being executed. Different techniques have been developed to overcome static analysis limitations. One of the most relevant is the **Function Call Monitoring.** Functions are fundamental building blocks of every program. They permit code reusing and a clear code structure when properly written. A method to understand software behavior is monitoring function calls: a hook function is triggered every time a monitoring function is called. Such hooks are responsible for the completion of required analysis functionality (e.g. a simple logging component). Hooking is the process of intercepting function calls. The main Windows System hooking points are:

- **Application Programming Interface**. Even simple program may make heavy use of operating system: system can execute thousands of system calls without the programmer's knowledge. In order to design a program, developers need to master the application programming interface (API) for the target systems, that is the set of functions available to an application programmer with a detailed semantics about passed parameters and returned values. APIs are accessed through libraries of code provided by the operating system.
- **System Calls**. A system call is a controlled entry point into the kernel, allowing a process to request that the kernel performs a service made available by an operating system. While invoking a system call could seem similar to a C function call, a system call performs some privileged actions as changing the processor state from user mode to kernel mode or specifying information to transfer between user and kernel mode. The system call interface is the run-time support of the system, it works as a bridge from API and system calls. Figure 1 shows the relationship between system calls, system call interface and API. Like benign process, malicious code running in user-space needs to invoke system calls: in this scenario hooking the system call interface points out every interaction within the system.

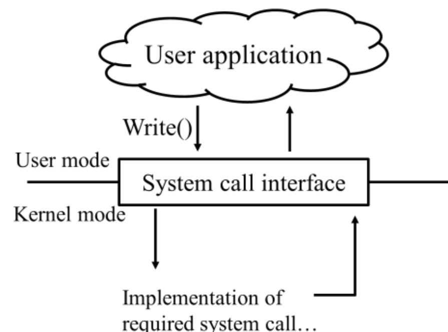

Figure 1: Handling of user application invoking a system call.

- **Windows Native API**. Windows Native API resides above system call interface and below Windows API. Contrarily to Windows API, this interface is stable only at service pack level and it can change over different updates. Native API gives an abstract view over system calls, managing all needed pre and post-processing operations. Such API allows deep control over the system calls, requiring a specific knowledge for relative system version.

A long term study is reported by Kharraz et al. (Kharraz et al., DIMVA, 2015). They analyze more than a thousand samples covering main ransomware families seen from 2006 to 2014. They show how malware interacts within a system and various strategy to mitigate it. Specifically they suggest monitoring API calls or File System activities and using decoy resources as additional measure. Many approaches have been proposed to analyze and detect malware intrusion: usually combining heuristics with behavior analysis to evade countermeasures (Dinaburg, CCS, 2008; Moser et al., S&P, 2007). Scaife et al. propose a pattern-recognition method to mitigate ransomware attacks, through monitoring I/O requests, (Scaife et al., ICDCS, 2016). Despite they claim to detect all tested malware samples, few files result encrypted during their experiments. Similarly Kharraz et al (Kharraz et al., Usenix, 2016) presented an automated approach to generate an artificial user environment and monitor filesystem activity. They add image analysis methods to detect typical ransom demand message. Other researchers included machine learning techniques to mitigate attacks from new family samples (Anderson et al., JCV, 2011; Kolter et al., JMLR, 2006; Rieck et al., JCS, 2011; Sgandurra et al., CoRR, 2016). The mentioned approaches suffer of high cost of misclassification errors. Ransomware detection do not permit false negative even minimally: compared to other domains, machine learning techniques are not well suitable to intrusion detection (Sommer et al., S&P, 2010). Continella et al. (Continella et al., ACSAC, 2016) try to mask previous issue introducing an automatic backup system. Except for few scenarios, they only introduced a work around. Finally Microsoft recently announced the upcoming new feature within the Windows Defender Antivirus called "*Controlled folder access*", able to monitor the changes that Apps make to files in certain protected folders. If an App attempts to make a change to these files, and the App is blacklisted by the new feature, you will get a notification about the attempt. The few information available seem to suggest an antivirus-like implementation that, as we will see, relies on a software layer higher than the one we implemented and thus much exposed to possible bypass.

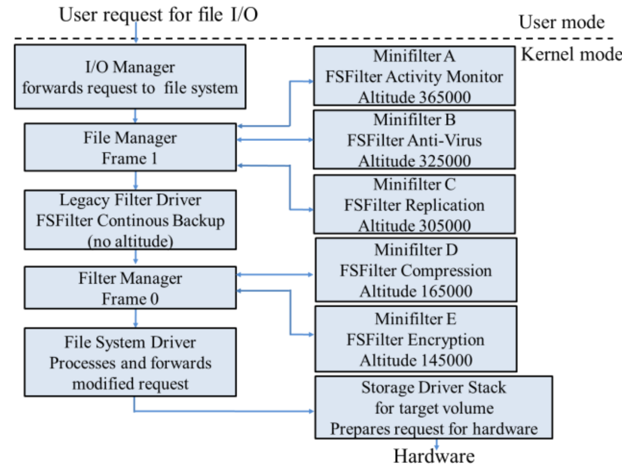

Figure 2: simplified I/O stack with the filter manager and three Minifilter drivers

3 The proposed Minifilter Driver

Windows documentation discourage from creating a new File system. Conversely it suggests using File System Driver and File System Filter Drivers to add new functionalities to existing File systems. File System Filter Drivers can intercept requests to a File system driver. In such a way it is possible to log, modify or prevent I/O operations.

Filter Manager is a kernel-mode driver adopting legacy file system filter model. It exposes a rich set of functionalities required for a File system filter driver. Writing a Minifilter driver consists to adopt these functions instead of implementing a legacy filter driver from scratch.

The Filter Manager is active only when a Minifilter driver is loaded in the operating system. Minifilter driver is indirectly attached to the file system stack for a target volume, registering its I/O operations to the Filter Manager. For each of them, a Minifilter driver can register functions to call at different times: before the beginning (e.g., pre-operation callback routine), after completion (e.g., post-operation callback routine) or both. Filter Manager is responsible to pass I/O Request Packets (IRPs) to Minifilter drivers with respect of a predefined order given by unique identifiers called **altitudes**. When a callback routine returns, it calls, in order, the next registered Minifilter callback routine. To guarantee interoperability with legacy filter drivers, more instances of Filter Manager, called **Frames**, can coexist in the file system stack to comply with the order of altitude assignment. A single Frame represents an interval of contiguous altitudes.

For example, an antivirus filter driver should be higher in the stack than a replication filter driver, so it can detect viruses and disinfect files before they are replicated to remote servers. Therefore, Filter drivers in the “File System Filter Anti-Virus group” are loaded before the ones in the “File System Filter-Replication-group”. Each load-order-group has a corresponding system-defined class and GUID class, used in the INF file for the filter driver.

A Minifilter driver’s altitude ensures that the instance of the Minifilter driver is always loaded at the appropriate location relative to other Minifilter driver instances, and it establishes the order in which the Filter Manager calls the Minifilter driver to handle I/O. *Altitudes are allocated and managed by Microsoft*. Figure 2 shows a simplified I/O stack with the Filter Manager and three Minifilter drivers.

For example, assuming all three Minifilter drivers in Figure 2 registered for the same I/O operation, the Filter Manager would call their pre-operation callback routines in order of altitude from highest to lowest (A, B, C), then forward the I/O request to the next-lower driver for further processing. When the Filter Manager receives the I/O request for completion, it calls each Minifilter driver’s post-operation callback routines in reverse order, from lowest to highest (C, B, A).

The specific Minifilter group we addressed is the one that includes Filter drivers that prevent the creation of specific files or file content, called “*FSFilter Content Screener*” whose altitude must be in the range 260000-269999. *Please note that the altitude of the group we addressed is lower than the one used for antiviruses (range 320000 – 329998).* Since the load order is regulated by the altitude, we can assume that the higher is the altitude the higher is the likelihood that a driver functionality can be compromised by lower drivers.

In this context, the development of a Minifilter Driver must necessarily be performed through the APIs and data structures provided by Microsoft. In general, the stages involved are:

- **Driver Entry Routine**. The system loads the driver automatically when DriverEntry is the entry point routine. Otherwise the developer must specify its name for the linker. In this case we adopted the given naming convention. Such routine is defined as:

```
NTSTATUS
(*PDRIVER_INITIALIZE) (
        IN  PDRIVER_OBJECT       DriverObject ,
        IN  PUNICODE_STRING     RegistryPath
) ;
```

  In this stage there are three main sub-stages to address, that are:

  - **Registry management**. Our driver global initialization consists in retrieving information from our Registry parameter keys that are provided through the .INF file for creating Registry keys, and including, among others a registry key called "Extensions" containing all the file types to

protect. Each one of these extensions are then used by the system to retrieve registry values containing default executable for filtered operations. Specifically each entry only contains a reference to another registry key.

- **Minifilter registration**. A Minifilter driver needs to be registered to the Filter Manager. Such task is completed in the FltRegisterFilter routine also passing its required callback routines and other driver information.

```
NTSTATUS    FltRegisterFilter (
  _In_              PDRIVER_OBJECT              Driver ,
  _In_    const     FLT_REGISTRATION            *Registration ,
_Out_               PFLT_FILTER                 *RetFilter
) ;
```

- **Minifilter filtering**. Minifilter driver only needs to call FltStartFiltering routine to start filtering. Such method is really simple and only takes as argument the filter pointer returned by FltRegisterFilter. It is not possible to start filtering before registering the Minifilter.

- **Callback Routines**. All interesting callback routines are passed together, as an array of FLT_OPERATION_REGISTRATION structures, to the filter manager. We do not use post-operation callback, so we explain in this section only pre-operation callback routine. It is defined as:

```
typedef    FLT_PREOP_CALLBACK_STATUS
           (*PFLT_PRE_OPERATION_CALLBACK) (
                  IN OUT  PFLT_CALLBACK_DATA          Data ,
                  IN      PCFLT_RELATED_OBJECTS       FltObjects ,
                  OUT     PVOID                       *CompletionContext
) ;
```

  We recognize two different patterns: Default Programs request access to relative files against non-default Programs. Taking such simplified approach, it is possible to build a bullet-proof system defense. Our Minifilter driver records a read request as pre-operation callback routine, implementing three tasks: check the requested file extension, extract the process name and compare it with the related default Program for granting the access.

In order to prevent the bypass of our Minifilter simply by changing the file extension, we classified as unsafe all the renaming of the file extensions as well as the file-read requests from programs other than the Default Programs.

4 Experiments

Malware analysis can be simply performed through *Cuckoo Sandbox* (Cuckoo Foundation, cuckoosandbox.org, 2016), an open source project, widely used in such a field, integrating an automated submission system, anti-detection modules and automated user-interaction; it seems to be perfect fit to our problem. Despite such a good premises, performing few tests revealed that many malware samples do not expose their malicious behavior although they have already shown it in a not-sandboxed environment. Rossow et al. (Rossow et al., IEEE Symp. on S&P, 2012) shows guidelines to design and present scientifically rigorous experiments. They gather common pitfalls in four main groups:

- **correct datasets** - choose correctly which samples should be in;
- **transparency** - give all needed information to understand and replicate experiments;
- **realism** - make experiments as real as possible;

- **safety** - do not be dangerous to others.

To test our driver we used VirtualBox 5.1 to setup a set of virtual machines containing different Windows versions: Windows 7 32-bit, Windows 8.1 32-bit and Windows 10 64-bit. We chose such configuration based on Desktop Operating System Market Share showed in Table 2. We tested our virtual environment against *Paranoid Fish*, the tool that employs several techniques to detect sandboxes and analysis environments as malware families do (Ortega, Paranoid Fish, 2016). For instance, it checks hardware limitations, debug-mode execution, virtualized environment detection (e.g., traces of Wine, VirtualBox or VMware), hooks detection and sandbox detection. An example of common Paranoid Fish output is given in Figure 3.

```
[ pafish ] Start
[ pafish ] Windows version: 5.1 build 2600
[ pafish ] CPU: GenuineIntel (HV: @) Intel(R) Core(TM) i7-4700MQ CPU @ 2.40 GHz
[ pafish ] CPU VM traced by checking the difference between CPU timestamp counters (rdtsc)
[ pafish ] Sandbox traced using mouse activity
[ pafish ] Sandbox traced by checking disk size <= 60GB via DeviceIoControl ()
[ pafish ] Sandbox traced by checking disk size <= 60GB via GetDiskFreeSpaceExA ()
[ pafish ] Sandbox traced by checking if NumberofProcessors is less than 2 via raw access
[ pafish ] Sandbox traced by checking if NumberofProcessors is less than 2 via GetSystemInfo ()
[ pafish ] Sandbox traced by checking if physical memory is less than 1Gb
```

Figure 3: Paranoid Fish output on a generic virtual environment

In order to make our laboratory as real as possible, we replaced default hardware components installed in virtual machine with our real components. We used an open source tool to address such tasks (Keri, WMI detection, 2016), performed through two scripts: a bash script to modify such information on the bare virtual machine and a powershell script to clean up VirtualBox residual information after Windows installation. Not having a wide compatibility, a custom script is sometime needed to fix minor issues, as in our case. After installing Windows system, the following steps are required to hide the virtual machine environment. We disabled the following services:

- **Windows Defender** - we don't want Windows real-time protection analyzing, and possibly blocking, our malware sample;
- **Windows Firewall** - although it may identify a testing environment, some malicious codes are inhibited when system firewall is activated;
- **Windows Update** - new updates may modify our configuration and may give an additional protection to the system;
- **Address Space Layout Randomization** - a feature that partially randomizes address space from buffer overflow attacks;
- **No eXecute technology** - another protection feature for specifying areas of memory that cannot be used for execution.

Hence, running the previous generated powershell script, we cleaned the registry keys, changing them to real system information, and performed some common tasks to mimic a basic user interaction (i.e. change user and computer name, create and delete some files). Further, we installed runtime libraries for Visual C and .NET, that are often used by malware, and common utility as Adobe Reader, Adobe Flash Player, VLC media player, Mozilla Firefox, OpenOffice and 7zip. Then we populate web browsers with a dump of typical user data, including fake credentials and browser history. Finally, we developed a Python script to populate our system (Hop and Fork, 2016), an easy-to-use open source software to create artificially directories and files. It takes as arguments the number of directories and files, the file extension list and the average file size. Randomly it generates contents and creates required files with given extensions. As a last step we wait almost 6 hours before saving virtual machine state, as it looks in a real working-section time. Taking that there is nothing more unreal than a system without user interaction, we developed another Python script that interacts with the system mimicking user behavior (e.g., open browsers and store a web page on the system, continually move cursor over the screen, etc.). After reproducing a real environment we took a

snapshot, that is a VirtualBox feature to save the current state of the entire virtual machine. In this way, each experiment can begin exactly in the same initial state. After granting the access to world-wide researcher repository of VirusShare, we downloaded a ransomware collection composed by about 36K elements. Monitoring activities was the most difficult task to cover. The Cuckoo Sandbox lacks on those tasks, revealing its virtual environment. Since we are only interested in ransomware encryption activities, we just checked files and desktop background: usually a ransom message is shown after encrypting data. Two different strategies have been used to analyze malware sample. In early stage we only collected desktop screenshots, a VirtualBox feature, every 30 seconds for 45 min per sample. In such a way it was possible to monitor changes over the virtual machine without any hook. Despite being an unusual technique, it resulted extremely effective, without revealing any monitoring scheme. Once collected our active dataset, we tested the environment by installing our driver. In this case there was no need of extra monitoring features: we manually checked the driver output logs and the decoy-files, looking for changes.

Type	Samples
Crypto ransomware	111
locker ransomware	30
fake antivirus	30
other active malware	37
others	819
Total samples	1027

Table 3. Classification of used dataset.

Table 3 shows the results of dataset classification. Despite the entire data set belongs to a ransomware repository publicly available, most of the samples resulted not compliant to our experimentation. In particular, almost 80% of total samples has not manifested malicious behavior, including a small set (17% of total samples) terminating with error messages. Only 10.8% of analyzed malware exhibited the typical crypto ransomware behavior. A second experiment has tested our driver against this threat: Table 4 shows the results. During ransomware attacks, our driver correctly prevented access to private data and *no valuable data have been encrypted*. From the malware point of view, errors generated during files interaction were managed in the simplest way, that is by skipping those files. Taking advantage of such generic exception handling, we have inhibited malware from running over protected files without being detected. In particular, none of analyzed samples changed its behavior due to our driver presence, this is an encouraging step to continue implementing solution directly in kernel mode. We do not discuss false positive cases, since our strategy classifies as suspicious all the software not included in whitelist. We believe that this is not a limitation, rather a strength for a system requiring strong security measures.

Evaluation	Results
total ransomware	111
detection rate	100%
false negative	0.0%
encrypted valuable data	0.0%

Table 4. Experimental results.

5 Conclusions and future work

There are limitations that we came across when developing our driver and we are perfectly aware that some work still needs to be done. The main limitations are the likelihood of false positives and

the driver's lack of flexibility. In the specific, the whitelist is statically configured during the initiation phase therefore it does not adapt to new demands or eventual user's habit change. Despite this solution fits perfectly in enterprise environments and in POS, ATM and Kiosk Mode Environments, problems may arise in home environments. However, the lack of flexibility should not be misunderstood as a lack of feasibility. On the other hand we believe that focusing on the protection of the user's valuable data as close as possible to the storage, is the only way to prevent ransomware attacks. Moreover, even if a kernel-mode approach might seem daunting, because a poorly written code can cause the entire computer to crash unexpectedly, it could ensure, at least in theory, better performances. In particular, the Filter Driver has been developed to trigger only when the file extension matches the white list, in order to be unperceived by home users or by systems not specifically oriented to high performance computing.

As said previously, evading a security measure implemented through a File System Filter Driver requires, generally, the development of a Filter Driver whose altitude is equal or less the one of the proposed solution. In addition, beyond the required skills (a File System Filter Driver is not a common executable), the correct installation of a Filter Driver on x64-based systems (starting from Windows Vista) requires administrative privileges and the .SYS file must be signed.

Hence, while not excluding a possible evolution of the threat, the current trend highlights that the majority of ransomware launches relatively straight-forward attack payloads with a very distinct and predictable behavior (Kharraz et al., DIMVA, 2015).

References

Anderson et al. (November 2011). *Graph-based malware detection using dynamic analysis.* In: Journal in Computer Virology, November 2011, Vol. 7, Issue 4, pp 247–258

Choi et al. (July 2016). *Ransomware Against Police: Diagnosis of Risk Factors via Application of Cyber-Routine Activities Theory.* In: International Journal of Forensic Science & Pathology, Vol. 4, July 2016, pp. 253-258.

Continella et al. (December 2016). *ShieldFS: a self-healing, ransomware-aware filesystem.* In: Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 16, Los Angeles, CA, USA, December 05-08, 2016, pp 336-347.

Cuckoo Foundation (2016). *Cuckoo Sanbox: Automated Malware Analysis*. 2016. Retrieved from https://cuckoosandbox.org.

Dinaburg et al. (October 2008). *Ether: malware analysis via hardware virtualization extensions.* In: Proceedings of the 15th ACM conference on Computer and communications security, CCS, Alexandria, Virginia, USA, October 27-31, 2008, pp 51-62.

Dingledine et al. (2004). *Tor: The second-generation onion router.* In: Proceedings of The 13th Usenix Security Symposium, San Diego, CA, August 09-13, 2004.

Egele et al. (February 2012). *A survey on automated dynamic malware-analysis techniques and tools.* In: Journal ACM Computing Surveys (CSUR), Vol. 44, Issue 2, February 2012, Article No. 6.

Forand (March 2015). *PARCC postponed as N.J. school district's network 'held hostage' for bitcoins.* Retrieved from http://www.nj.com/gloucester-county/index.ssf/2015/03/nj_school_districts_network_held_hostage_for_500_i.html

Furnell et al. (November 2012). *Power to the people? The evolving recognition of human aspects of security.* In: Computers & Security, Vol. 31, Issue 8, November 2012, pp 983-988.

Gazet (February 2010). *Comparative analysis of various ransomware virii*. In Journal in Computer Virology, Vol. 6, Issue 1, february 2010, pp 77–90.

Green (July 2016). *Hospitals are hit with 88% of all ransomware attacks.* Retrieved from http://www.beckershospitalreview.com/healthcare-information-technology/hospitals-are-hit-with-88-of-all-ransomware-attacks.html.

Hop and Fork (2016). *vm-palm-tree*. Retrieved from https://github.com/hopandfork/vm-palm-tree.

Keri (2016). *WMI detection prevented*. Retrieved from https://github.com/nsmfoo/antivmdetection

Kharraz et al. (July 2015). *Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks.* In: Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, (DIMVA), Milan, Italy, July 09-10, 2015, Vol. 9148, pp. 3-24.

Kharraz et al. (August 2016). *UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware.* In: Proceedings of the 25th USENIX Security Symposium, Austin, TX, USA, August 10-12 2016, pp. 757-772.

Kolter et al. (December 2006). *Learning to detect and classify malicious executables in the wild.* In: The Journal of Machine Learning Research, Vol. 7, pp 2721-2744.

Moser et al. (May 2007). *Exploring Multiple Execution Paths for Malware Analysis.* In: Proceedings of the 2007 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, May 20-23, 2007, pp 231-245.

Moser et al. (December 2007). *Limits of static analysis for malware detection.* In: Proceedings of the 23rd Annual Computer Security Applications Conference, ACSAC, Miami Beach, FL, USA, 10-14 Dec 2007.

Nakamoto (2008). *Bitcoin: A peer-to-peer electronic cash system*. Retrieved from https://bitcoin.org/bitcoin.pdf

Ortega (2016). *Paranoid Fish.* Retrieved from https://github.com/a0rtega/pafish

Osterman Research (August, 2016). *Understanding the Depth of the Global Ransomware Problem*. Retrieved from https://www.malwarebytes.com/surveys/ransomware/?aliId=13242065

Rieck et al. (December 2011). *Automatic analysis of malware behavior using machine learning.* In: Journal of Computer Security, Vol. 19, Issue 4, December 2011, pp 639-668.

Rossow et al. (May 2012). *Prudent Practices for Designing Malware Experiments: Status Quo and Outlook.* In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP 12, San Francisco, CA, USA, 20-23 May 2012, pp 65-79.

Savage et al. (August, 2015). *The evolution of ransomware.* Retrieved from http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-evolution-of-ransomware.pdf

Scaife et al. (June 2016). *CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data*. In: Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS), Nara, Japan, 27-30 June 2016.

Sgandurra et al. (2016). *Automated Dynamic Analysis of Ransomware: Benefits, Limitations and use for Detection.* In: ArXiv e-prints, arXiv:1609.03020

Sommer et al. (May 2010). *Outside the Closed World: On Using Machine Learning for Network Intrusion Detection.* In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 10, Berkeley/Oakland, CA, USA, May 16-19, 2010, pp 305-316.

SonicWall (2017). *SonicWall Annual Threat Report. SonicWall*. Retrieved from https://www.sonicwall.com/whitepaper/2017-sonicwall-annual-threat-report8121810/

Stone-Gross et al. (2013). *The Underground Economy of Fake Antivirus Software*. In: Schneier B. (eds) Economics of Information Security and Privacy III. Springer, New York, NY.

Symantec Corporation (October 2009). *Symantec Report on Rogue Security Software.* Retrieved from http://eval.symantec.com/mktginfo/enterprise/white_papers/b-symc_report_on_rogue_security_software_WP_20100385.en-us.pdf.

Symantec Corporation (April, 2016). *Internet Security Threat Report.* Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf

Symantec Corporation (August 2016). *Ransomware and Businesses 2016.* Retrieved from https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ISTR2016_Ransomware_and_Businesses.pdf