Malware Detection in Secured Systems

Claes Adam Wendelin
Betreuer: Stefan Liebald, M. Sc.
Seminar Future Internet SS2017
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: wendelin@in.tum.de

ABSTRACT

This paper presents the state-of-the-art in malware detection and what properties of malware they depend on. Understanding the connection between different malware and detection methods is required for further research and design of tomorrow's malware detection. The value of information is big today considering trade secrets and the amount of personal data available on our computer systems.

Three analysis methods are covered. Static analysis detects simple malware well, but has known flaws that can be used to circumvent detection. Dynamic analysis with its focus on recognizing behavior has robustness and generalization, but raises questions regarding the future complexity of the analysis. Automating malware detection with machine learning shows potential although the current techniques suffer from a high false positive rate.

The kernel is a source of trust in malware detection, which testaments the importance of its integrity. Preventing malware from hijacking the kernel can be done with signed drivers or protection of function pointers.

Keywords

Malware Detection, Static Analysis, Dynamic Analysis, Heuristic Analysis, Kernel Integrity

1. INTRODUCTION

Malicious software (malware) is software designed to cause unwanted behavior on a computer. Incentives to write and deploy malware can be economic, cultural, social, or politic [20], which can affect vital parts of society such as healthcare, finance and trust. The United Kingdom government estimated in 2011 that the overall cost of cyber crime to its economy amounts to £27 billion [12].

A malware infected computer can be used as a resource to spread the malware further or provide services such as spam. A specific network of infected computers, Cutwail, was reported by Symantec [34] to be responsible for 46.5% of spam on the Internet and estimated by [43] to have made profits from $1.7 million to $4.2 million in the time period 2009-2011.

Malware can also be used to gain unauthorized access to information. Trade secrets are an important economic asset to companies. Hiding information instead of making it generally known can give an advantage to businesses, but a trade secret has no exclusive rights compared to a patent, which makes them a valuable target for cyber espionage. [38]

This paper reviews malware detection techniques and what aspects of malware they depend on. Background to the requirement of sophisticated detection methods is covered in Section 2. Two system penetration techniques are listed in Section 3 and followed by detection methods in Section 4. Various ways to protect kernel integrity is found in Section 5. Finally the methods are assessed and concluded in Section 6 and 7 respectively.

2. BACKGROUND

Advanced Persistent Threats (APT) are malware designed to target a specific organization with often custom written malware not yet recognized by anti-malware detection [7]. Thomson [47] addresses the risk of APTs and the difficulty of detecting them due to their specialized nature. An APT has long term goals and propagates slowly and carefully to get to its objective, which is greatly contrasted by most traditional malware that operates on a hit-and-run basis [8]. Anti-malware software has been developed to focus on the common malware that tries to spread and act as fast as possible to infect the widest possible target group before being analyzed and protected against. The meticulous nature of APT in addition to its narrow target group reduces the chance of it being discovered and analyzed by commodity anti-malware. More advanced detection methods are therefore needed to combat the APTs.

An important part of APTs lies in the preparation for the attack. The target group and system must be studied closely to enable system penetration and avoiding detection. System penetration methods are discussed further in Section 3 and is the first real contact between the target and the malware. The exploit can be launched using a small infection file, that after compromising the system and stealing valid system credentials, downloads the real malware modules and then erases itself to hide the intrusion [7]. The APT is then on the system with valid user access and no direct connection with the original infection point.

The custom written nature of the APT means that anti-malware has no record of it and therefore no immediate reason to suspect its files. Analyzing the behavior is also a challenge as it attempts to mirror user behavior to prevent its valid access from getting revoked. While covert on the system, the APT tries to get access to the desired data,

doi: 10.2313/NET-2017-09-1_11

which is to be exported outside the system for collection.

3. SYSTEM PENETRATION

The first step of malware propagation is to infect a host with the malware. There exists a wide variety of methods to distribute malware, but the two that will be covered here are social engineering and quantum insert. Both have in common that they can be used for malware attacks against specific targets, a desirable trait for APTs.

3.1 Social Engineering

Social engineering is a mix of psychology, human nature and manipulation. It is used to manipulate people to do something desirable for you even though it might be undesirable for them. Social engineering is comparable to a magician, which with subtle cues masterfully diverts your attention to whatever they want while they perform the magic.

Technical countermeasures to malware are useless if a user can be made to bypass them. People are often called the weakest link in the security system and it is their naivety and trust that social engineering methods try to exploit. A simple trust in a friend sending you a link via social media can be enough to compromise a system. The botnet Koobface [46] has reportedly access to over 1800 compromised hosts, access which was gained by spamming enticing links on Twitter and Facebook. Infected accounts become part of the spreading as they are used by the botnet to further reach out to their friends with links.

Emotions are an important part of social engineering. Instilling feelings such as fear, greed, sympathy or anger can make us do things we otherwise would not do and have been abused since the beginning of time. Abraham et al. [1] summarizes social engineering malware principles and draws the conclusion that to combat social engineering - purely technical solutions are not enough. They suggest improvements by raising awareness, further monitoring, security policies and motivating users to follow secure practices.

Phishing is a social engineering technique where the user is tricked into giving away sensitive information, often through e-mails or social media, by disguising itself as a legitimate source [45]. According to the Anti-Phishing Working Group, the number of phishing attempts increased by 65% from 2015 to 2016 yielding a total of 1,220,523 attempts in 2016 [3].

3.2 Quantum Insert

Quantum insert is a technical approach to implant malware on a targeted set of users by utilizing an inherent flaw in TCP. *The Intercept* has reported usage of Quantum Insert by the governmental intelligence bureaus NSA [19] and GCHQ [18] to facilitate unauthorized surveillance and monitoring of traffic.

When a desired target opens up a connection with a website, the attacker snoops on the TCP packet containing the HTTP request and quickly spoofs an answer packet that would redirect the target to an infected website instead [22]. All required information to create a valid answer packet exists in the outgoing packet, so the target's system will accept the first one that reaches it. This puts requirements on the attack, since the attacker's server must be in close vicinity to the target to get a minimal latency.

Performing an attack like this requires processing of huge amounts of data in real-time to find interesting users and a nearby server infrastructure that can send the spoofed packet and serve a malicious website. An infection similar to Quantum Insert is China's Great Cannon [29], which intercepts and injects malicious Javascript code into packets with a set of target addresses.

Detecting Quantum Insert is not straightforward, but [22] suggests multiple symptoms, which can indicate an ongoing attack: (1) duplicate TCP packets with different payloads, (2) anomalies in Time-To-Live values due to different hop counts in the routes and (3) other inconsistent values in the TCP header. Although duplicate TCP packets could indicate Quantum Insert, being too considerate with duplicates might make the system significantly weaker to denial-of-service attacks using flooding [33].

4. DETECTION METHODS

The following sections will reason about detection methods used by anti-malware and various countermeasures by the malware to avoid them. An aspect of the malware that can be used for identifying it as such is called a feature. The detection methods focus on different parts of the malware to extract features from.

There are two ways in general to analyze malware. Dynamic analysis, which is described in Section 4.2, and static analysis described below.

4.1 Static Analysis

Static analysis is done without running any of the malware's code. The malware therefore will not be executed by the computer unless deemed safe by the analysis. A common technique used in static analysis is syntactic signature detection, but alternative detection methods with semantic signatures and heuristics have been proposed.

A first step before the malware code can be deconstructed and analyzed, is unpacking and/or decryption [35]. Malware uses compression to reduce the size of their executable, which causes a lower impact transfer of the malware to the to be infected system. Unpacking a file requires knowledge of the packing method, which means that anti-malware software can not always unpack them. Denying packed files altogether is not a feasible solution, since legitimate software also utilizes packing.

The simplest version of malware utilizing packing would consist of a malign binary, which is the payload, and decryption code that can unpack the malign code and run it. Since in such malware, the decryption code is constant, anti-malware can analyze the decryptor and recognize malware if it has been detected before [57]. Another option to combat packing is to let the malware unpack itself and analyze its code in memory using dynamic analysis, which is covered in Section 4.2.

4.1.1 Syntactic Signature Detection

doi: 10.2313/NET-2017-09-1_11

This technique utilizes that malicious instructions or code sequences of a known malware can be used to detect the same malware again. Mapping an arbitrary length of data to a fixed size can be done using a hash function. Important aspects of the hash function is that it is deterministic to always produce the same output for the same input, and that it evenly distributes the input data over the output to reduce the risk of collisions. By calculating a hash over the malign code, you get a syntactic signature. Anti-malware software collects known malware's syntactic signatures in a database, which allows them to be detected in the future. A syntactic signature detector is dependent on the database to detect malware, which makes it prone to new or changed malware.

Malware can then be detected by using regular expressions or pattern matching to compare a file's content to the signature database. A matching signature means that malicious code has been spotted in the scanned file. Finding malicious behavior like this is highly dependent on a consistent syntax of the code, since different, misplaced or additional instructions will change the hash output. Robustness against syntax changes can be improved by analyzing the file's content in chunks to generate a series of signatures [41]. A percentage of matching signatures can be enough to recognize malware.

Syntactic signature detection in static analysis has two major malware types that it has problems detecting: metamorphic malware and polymorphic malware. Their common denominator is their ability to constantly transform their syntactic signature to avoid being categorized by a syntactic signature database.

Polymorphic malware consists of an encryptor, decryptor and a constant payload that it tries to hide from the signature detector. The weakness of the earlier mentioned packing is that it is constant, so once it is part of the signature database, it can be detected again. Polymorphic malware gets rid of that weakness by mutating its static payload and decryptor. For every propagation, the polymorphic malware generates a new encryption key, which the encryptor uses to pack the static binary. Using a different key every encryption together with a different decryptor ensures that all copies of the malware have a different syntactic signature. [15]

The decryptor is generated by a polymorphic engine using various code obfuscation techniques to make it harder to analyze and different from previous versions. Different techniques include[9, 57]:

- Nop-insertion, the addition of no operation instructions.
- Code transposition, switching the order of independent instructions.
- Register reassignment, changing the used registers for instructions.
- Code substitution, exchanging instructions with semantically equivalent ones.

Creating an polymorphic engine that consistently produces distinct copies without being recognizable is usually more sophisticated than the malware itself, which can manifest in flaws that aid in its detection[15, 26].

Polymorphic malware does not mutate its behavior and instead hides it with encryption. A malware type heavily involved in obfuscation rather than encryption is the metamorphic malware, which does not need encryption to stay hidden. Instead, it fundamentally changes its code every propagation to always have different syntactic signatures [15]. The metamorphic malware is transformed using the same code obfuscation techniques as the polymorphic malware, but the scope is increased to the whole malware.

The process of metamorphic malware's mutation can include up to five steps [37, 35].

Firstly, the malware must decode its instructions into an intermediate form that describes its functionality. Combinations of different instructions can be functionally equivalent when combining certain arguments, which makes the extraction a complex task.

Secondly, the metamorphic engine shrinks the intermediate form by removing instructions that does not do anything and was included by the previous mutation. Without this step, metamorphic malware could gain size every mutation and therefore increasing its workload every iteration. Keeping the final code a similar size is important to avoid giving anti-malware features that can be tracked.

Thirdly, the intermediate form's control flow is changed by reordering instructions and submodules, which are linked together with jump instructions. The metamorphic engine must analyze which instructions are independent from each other to retain the functionality while transforming as much as possible.

Fourthly, no-operation instructions and other functionally irrelevant instructions are added to expand the malware once more. This addition makes sure that syntactic signature detection of code sections with low permutability is improbable.

Finally, the metamorphic engine reassembles the malware's intermediate form back into instructions for the infected system.

The metamorphic engine tries to make the code as hard as possible to analyze, which further makes the transformation of itself more complicated. Because of an increased amount of instructions from the imperfect obfuscation, the CPU workload is also increased. Anti malware software can monitor CPU idle time and set benchmarks, which can lead to detection of a malware's workload signature [35].

4.1.2 Semantic Signature Detection

Malware detection methods focused on analyzing the syntactic nature of the malware have weaknesses toward code obfuscation. Semantic signature detection avoids those flaws by focusing on the behavior of the program. Variants of the malware, that would possibly require different syntactic sig-

doi: 10.2313/NET-2017-09-1_11

natures, all have a similar functional flow [24]. Multiple variants of the same malware can therefore be detected using just one semantic signature.

Semantic signature detection is being explored as a valid alternative to syntactic signature detection for static detection. Christodorescu et al. [11] introduces formal semantics to classify malicious behavior.

Formal semantics acts as a specification language with which you can define behavior using a template. Templates consist of a sequence of instructions, variables and symbolic constants. Specific registers and values in the malware's code are abstracted away in the template, but their dependencies are preserved. The templates are therefore not affected by common code obfuscation techniques such as register reassignment and no-operation instructions. Malicious behavior is specified in templates and compared to the dubious binary to determine if it is malicious or not.

A similar semantic approach has been explored in [24], where Kinder et al. uses model checking to verify malware specifications. Model checking is a way to verify the correctness of a finite-state system against a specification (e.g. only one process allowed in a critical section). In this case, the system is verified against unwanted behavior, which is defined as a set of states, transitions and labels. The malware binary is disassembled into machine instructions that are used to build a model of the program, which is verified with model checking against specified malware behavior.

Both of these semantic analyses try to construct a control flow of the target binary to determine whether it is malign or benign. By making the static analysis harder using binary obfuscation, malware could avoid detection. Moser et al. [32] point out that a vital part of semantic analysis lies in establishing relationships between constants. Constants in the binary could be used for arithmetic operations or as jump destinations. Obfuscating jump destinations can make control flow analyses miss the malicious set of operations that would otherwise be uncovered. Obfuscating binary constants using the known hard to solve 3-satisfiability problem can make the derivation of the constants in static analysis an NP-hard problem [32].

4.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis actually executes the malware and from that tries to deduce if its malign or not. There are primarily two ways to dynamically analyze malware that are used by anti-malware software. Either of them uses emulation to execute the malware in a safe environment to hopefully reveal its intent or monitor a program's behavior in real time to detect and prevent suspicious actions before executed [2]. The most common analysis method used for dynamic analysis is with behavioral traits on the basis that malware is classified by what it does instead of what it looks like. As system calls are the interface to the operating system, monitoring those can reveal a program's resources and actions [16]. Lots of papers can be found that utilizes system call monitoring in a dynamic analysis fashion, including but not limited to [21, 30, 10, 36, 25, 27]. Network traffic can also be used to characterize malware [21].

Dynamic analysis is inherently resistant to the code obfuscations mentioned in Section 4.1 that tries to hinder static analysis. As the actually executed instructions are considered in dynamic analysis, obfuscated code can be identified after its decryption to valid instructions. For example, obfuscated constants [32] that hide jump addresses or data dependencies behind complex calculations will unwind themselves into proper instructions when executed, which makes them easy to analyze with dynamic analysis. Polymorphic malware is also easier to analyze with dynamic methods, since its decryptor will unpack its static payload into memory to let it run. Static payload is easily identified by syntactic signature detection.

A flaw of dynamic analysis is that only one execution path of the monitored program is revealed. Malware can exploit this by detecting that they are being monitored and hide malicious functionality or immediately exit. To do this, malware can either look for clues of specific analysis tools in files, registry or processes, or by analyzing imperfections that occur during emulation such as timing properties or variations of instructions [5].

Malware can not be analyzed forever, which means that *logic bombs*, malware that triggers on an event, can go undetected by dynamic analyzers. Logic bombs could trigger on a date, a user input or remote control via network [14]. The event that triggers the malware is often external, so tracking everything that the malware reads together with its control flow decisions can be used to simulate different inputs to the malware and reveal its multiple execution paths [31].

Ether [13] is a malware analyzer that uses hardware virtualization extensions to allow it to reside entirely outside the target operating system environment. Thereby no trace of the analyzer can be found in the emulation. Another analyzer trying to stay transparent to malware is Cobra [48]. Cobra disassembles and executes malware code in blocks that are scanned for instructions that might give away the environment. Such instructions are replaced with safe versions that will not give away the analysis tool. Both of these tools incur a significant performance overhead, which can be undesirable. Another technique developed for analysis-aware malware introduced in [5] compares the execution of malware in an analysis environment with execution in a reference environment. If the execution differs, the malware should be analyzed more thorough.

4.2.1 Semantic Signature Detection

Semantic signature detection tries classify malware by creating malicious behavior signatures. The state of the art in semantic signature detection is using a behavior graph, with which malware detection is an NP-complete problem [17]. Behavior graphs are composed out of nodes, which represent actions (e.g. system calls), and edges that represent data dependencies between nodes. Behavior graphs can be used as a detection method by monitoring an unknown program's system calls with their dependencies, and match that behavior with nodes in the graph. An enough matched graph indicates malicious behavior [25].

On an end-system, the semantic signature would be detected after the malicious activity has been performed, which puts

a clock on the anti-malware to prevent the malware from doing further damage and repair what was done. A too slow detection system might give the malware enough time to complete its objective. The complexity of behavior graphs has lead to the development of a lightweight alternative to behavioral graphs. Lu et al. [27] suggest a solution that generalizes system calls by classifying them based on the resource used (e.g. file, socket) and a classification of those resources based on similar functionality. The abstraction level is higher than a behavior graph, which makes it robust against code-obfuscation.

The current state of the art behavior graphs focus their analysis on a single process to reduce the complexity and computation power required. Ma et al. have shown [28] that multi process malware can automatically be generated from a malware's source code and successfully avoid semantic detection in single process based malware detector. The technique to generate multi process malware is shown in figure 1. The process P_0's two system calls are split up between two processes P_0' and P_1'. The state change caused by the system call is transferred between the processes so that the execution of P_0 is equivalent to the execution of P_0' and P_1'.

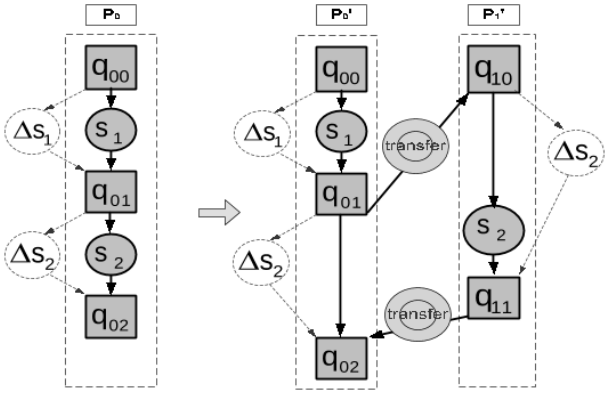

Figure 1: Creation of multi process malware from single process malware. Source: [28]

4.3 Heuristic Analysis

Heuristic methods make use of data mining and machine learning techniques to try and detect malware. Such techniques try to classify malware using knowledge from previous data that can either be labeled or not. The algorithm uses a feature vector as an input and outputs a classification of the supposed to be malware that is either malign or benign. A feature vector contains different information about the malware that is used in the classification process. The chosen features are vital for the success. Five features are discussed in [6]:

- API/System calls: a program's requests to the operating system.
- Control flow graphs: a program's statements and control flow to construct a directed graph.
- N-Grams: substrings of a program's binary code.
- OpCode: sequences and statistical frequencies of different machine language instructions.
- Hybrid features: a combination of feature types.

The feature vector is built from the chosen data. Classification based on system calls could use sequences of system calls as input, which the algorithm processes to determine if its an illicit sequence.

Yan et al. [55] have made a study of useful features in automated malware classification. Their results point out that the header of a Windows executable is highly discriminatory. But there is a dilemma, since an adoption of the technique might lead to feature obfuscation by malware writers. Robustness of features is therefore an important issue and further discussed in [56].

Machine learning techniques can be used to detect metamorphic malware. A detection technique using Hidden Markov Models is proposed in [54], where a model is trained on a family of metamorphic malware to be able to recognize its defining features. The model becomes specialized in one malware family, but managed to also detect some malware from another family. The same weakness of similarity is exploited by Runwal et al. [40] in a detection technique involving the similarity score between the opcode graphs of metamorphic families and normal files.

Heuristic analysis can be a very powerful technique considering its automation possibilities, but as [6] summarizes: heuristic malware detection suffers from a high false positive ratio.

5. PROTECTING KERNEL INTEGRITY

The kernel of an operating system has complete control of the system. It handles start-up, input and output from programs, peripherals, memory and CPU. Access to the kernel can be used to extend the operating system with malign functionality such as backdoors, logging keystrokes, escalating privilege of malware or tampering with other defense mechanisms. The premise of malware detection methods is that they have higher privilege than the malware, which is invalid if the malware has kernel access.

The malware type rootkit will try to stay covert after infection while it fulfills its illicit objective. A typical hiding method is to remove its entry from the system processes list. Kernel rootkits are malware that targets the kernel and actively tries to tamper with it.

Baliga et al. [4] propose a general classifying scheme for rootkit attacks, which are categorized by the tampering technique: *Control hijacking* and *control interception* both manipulate the control flow of the kernel by changing the system call table, interrupt descriptor table and kernel code. *Control tapping* will ensure that the attack code is executed on every invocation of a specific function without affecting it. *Data value manipulation* tries to indirectly manipulate the kernel by changing values of critical variables. *Inconsistent data structures* similarly tamper with kernel data structures, which are assumed to be consistent, but are made inconsistent and can be used to hide processes and modules. A rootkit can combine a variation of these to accomplish its task on the system.

doi: 10.2313/NET-2017-09-1_11

Defense against rootkits can be divided into prevention and detection. The goal of prevention is to avoid unauthorized access to the kernel by keeping its integrity consistent. A method of doing so is kernel module signing. Such a policy requires all kernel modules to have an embedded digital signature, which can be verified against a source of trust. Windows requires signed drivers by default [52] - a feature that only can be disabled via a reboot. As [23] mentions, kernel module signing security is made on the assumption that all loaded code in the kernel is safe and prevents execution of unsigned kernel code. The Turla (a.k.a. Uroburous, Snake) [44] rootkit managed to bypass Windows' driver signature policy by leveraging a flaw in the virtualization tool VirtualBox [49], which included a signed driver containing a privilege escalation vulnerability.

Detection on the other hand tries to safeguard the kernel integrity by detecting unexpected changes in critical regions. As function call hooking is often a vital part of rootkits, detecting and preventing the hijacking could protect the kernel's integrity. Detecting illegitimate function pointers (hooks) requires a reference to compare with. Some detectors [50, 58, 51, 39] dynamically find kernel hooks, which they use shadow memory to keep consistent. Shadow memory is a technique that maps memory bytes to metadata. The metadata could keep track of the number of writes to a specific memory section or also if the memory originates from a trusted source.

Verifying that only trusted sources modify function pointers could protect their integrity and prevent rootkits. Hardware-based memory protection can safeguard the function pointers, but it only protects on a page-level granularity. Data that does not need memory protection, but is on the same page as a function pointer, would get an access overhead using such a technique.

A novel approach by [50] is to generate a shadow copy of all function pointers in a centralized page-aligned memory location depicted in figure 2, where hardware-based protection is utilized efficiently. Function pointers are accessed through an indirection layer with either read or write operations. A read request will be fulfilled directly by the indirection layer and a write operation will be passed on to the hardware-based memory protection to validate the request.

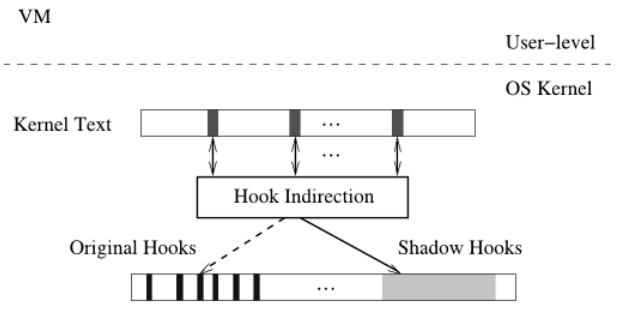

Figure 2: Redirection of function pointers to protect their integrity. Source[50]

Windows has an approach similar to shadow memory that saves data about kernel function pointers and interrupt tables to allow validation. The data can be known-good copies and metadata in the form of checksums, which allows the operating system to validate kernel integrity during run-time to prevent tampering [53, 42].

Although preserving kernel integrity thwarts the majority of rootkits, such a technique can be bypassed using return-oriented rootkits [23]. Return-oriented rootkits construct malicious stack frames by using *gadgets*. A *gadget* is a combination of kernel instructions that together form a well-defined behavior, e.g. computing AND of two operands. The self-contained *gadgets* can be combined to manipulate the stack and launch attacks upon the system. Integrity checks and memory protection fails since all malicious activity is done with safe instructions already in the kernel.

6. ASSESSMENT

Static analysis and syntactic signature detection methods are a great first defense against malware. Using a dictionary of previously known malware makes it harder for old malware to infect systems again, but it is inherently weak against new and advanced malware.

Semantic detection in static analysis is powerful due to its complete analysis of all the malware's execution paths and the generality of its signatures. Future malware using an already known way to achieve its objective can be detected without updating the signature database. The limited amount of system calls suggests that a thorough cataloging of malicious sequences are possible.

An issue with semantic detection arises with the increased complexity of analysis that obfuscation can yield. Most of the semantic techniques try to construct a behavior signatures from graphs, but correlation between system calls and actions are not always obvious. A malware split up into multiple processes requires a more advanced analysis to try find data or control dependencies between processes. The analysis becomes even harder if dependencies between processes are obfuscated by using a non-conventional messaging system. Process A and B could communicate with different servers, which relay messages between each other and thereby hide the connection between process A and B.

Dynamic analysis is unfazed by encryption methods and obfuscated constants, which testaments its relevancy besides static analysis. As of now, dynamic analysis can be efficient against traditional malware that acts on a rapid pace, but is lacking if the malware is patient. An APT that can hide and stay inactive for months is hard to detect using dynamic analysis as the malware does not generate any evidence against itself until it actually acts.

Protecting against advanced malware and especially new malware requires a general and automatic classification system that can perform without human intervention. Heuristic methods are not mature enough yet, but their success in other areas showcase their power in classification based on features, which makes it a promising research area. Another important area is kernel integrity, since a breached kernel can tamper with anti-malware and do substantial damage to the system.

doi: 10.2313/NET-2017-09-1_11

7. CONCLUSION

There exists a wide variety of malware detection methods, but none of them are robust to all malware. Different advantages and disadvantages in analysis methods suggests that a detector should not be limited to one method, but rather utilize a wide set of different tools to complement each other.

The possibilities to automatically increase the complexity of static analysis and dynamic analysis highlights the challenges that anti-malware software faces. Automatically detecting new malware is an important topic, where machine learning is a promising area. Machine learning has made breakthroughs the last years that will hopefully spread to malware detection.

8. REFERENCES

[1] S. Abraham and I. Chengalur-Smith. An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society*, 32(3):183–196, 2010.

[2] E. Al Daoud, I. H. Jebril, and B. Zaqaibeh. Computer virus strategies and detection methods. *Int. J. Open Problems Compt. Math*, 1(2):12–20, 2008.

[3] APWG. Phishing Activity Trends Report, 4th Quarter 2016. `https://docs.apwg.org/reports/apwg_trends_report_q4_2016.pdf`. Accessed: 2017-04-09.

[4] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 246–251. IEEE, 2007.

[5] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.

[6] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh. A survey on heuristic malware detection techniques. In *Information and Knowledge Technology (IKT), 2013 5th Conference on*, pages 113–120. IEEE, 2013.

[7] R. Brewer. Advanced persistent threats: minimising the damage. *Network security*, 2014(4):5–9, 2014.

[8] P. Chen, L. Desmet, and C. Huygens. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security*, pages 63–72. Springer, 2014.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.

[10] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[12] Detica and C. Office. The cost of cybercrime: A detica resport in partnership with the office of cyber security and information assurance in the cabinet office. Technical report, Cabinet Office and National security and intelligence, 2011.

[13] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[15] P. Ferrie and P. Ször. Hunting for metamorphic. *Virus, págs*, pages 123–143, 2001.

[16] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 120–128. IEEE, 1996.

[17] M. Fredrikson, M. Christodorescu, and S. Jha. Dynamic behavior matching: A complexity analysis and new approximation algorithms. In *International Conference on Automated Deduction*, pages 252–267. Springer, 2011.

[18] R. Gallagher. Operation socialist: The inside story of how british spies hacked belgium's largest telco. *The Intercept*, 12, 2014.

[19] R. Gallagher and G. Greenwald. How the nsa plans to infect 'millions' of computers with malware. *The Intercept*, 12, 2014.

[20] R. Gandhi, A. Sharma, W. Mahoney, W. Sousan, Q. Zhu, and P. Laplante. Dimensions of cyber-attacks: Cultural, social, economic, and political. *IEEE Technology and Society Magazine*, 30(1):28–38, 2011.

[21] A. R. Grégio, D. S. Fernandes Filho, V. M. Afonso, R. D. Santos, M. Jino, and P. L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. In *SPIE Defense, Security, and Sensing*, pages 80590O–80590O. International Society for Optics and Photonics, 2011.

[22] L. Haagsma. Deep dive into quantum insert. *Online at https://blog.fox-it.com/2015/04/20/deep-dive-into-quantum-insert*, 2015.

[23] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[24] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7(4):424–438, 2010.

[25] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, pages 351–366, 2009.

[26] E. Konstantinou and S. Wolthusen. Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15, 2008.

[27] H. Lu, B. Zhao, J. Su, and P. Xie. Generating lightweight behavioral signature for malware detection in people-centric sensing. *Wireless personal communications*, 75(3):1591–1609, 2014.

[28] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1):1–13, 2012.

[29] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. China's great cannon. *Citizen Lab*, 10, 2015.

[30] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.

[31] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[32] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[33] R. M. Needham. Denial of service. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 151–153. ACM, 1993.

[34] Nisbet. Cutwail Takedown Cripples Bredolab Trojan; No Effect on Spam Levels. `https://www.symantec.com/connect/blogs/cutwail-takedown-cripples-bredolab-trojan-no-effect-spam-levels`. Accessed: 2017-04-09.

[35] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[36] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[37] F. Perriot, P. Ször, and P. Ferrie. Striking similarites: Win32/simile and metamorphic virus code. *Symantec Corporation*, 2003.

[38] J. Pooley and D. P. Westman. Trade secrets. In *WIPO Magazine*. Law Journal Seminars-Press, 1997.

[39] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *International Workshop on Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2008.

[40] N. Runwal, R. M. Low, and M. Stamp. Opcode graph similarity and metamorphic detection. *Journal in Computer Virology*, 8(1-2):37–52, 2012.

[41] W. Scheirer and M. C. Chuah. Syntax vs. semantics: competing approaches to dynamic network intrusion detection. *International Journal of Security and Networks*, 3(1):24–35, 2008.

[42] Skywing. Patchguard reloaded: A brief analysis of patchguard version 3. *Uninformed Journal*, 8(5), Sept. 2007.

[43] B. Stone-Gross, T. Holz, G. Stringhini, and G. Vigna. The underground economy of spam: A botmaster's perspective of coordinating large-scale spam campaigns. *LEET*, 11:4–4, 2011.

[44] B. Systems. The Snake Campaign: Cyber Espionage Toolkit. `http://www.baesystems.com/en/cybersecurity/feature/the-snake-campaign`. Accessed: 2017-04-09 Last updated: 2016-01.

[45] C. Tankard. Advanced persistent threats and how to monitor and deter them. *Network security*, 2011(8):16–19, 2011.

[46] K. Thomas and D. M. Nicol. The koobface botnet and the rise of social malware. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 63–70. IEEE, 2010.

[47] G. Thomson. APTs: a poorly understood challenge. *Network Security*, 2011(11):9–11, 2011.

[48] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[49] VirtualBox. `https://www.virtualbox.org/`.

[50] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 545–554. ACM, 2009.

[51] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *International Workshop on Recent Advances in Intrusion Detection*, pages 21–38. Springer, 2008.

[52] Windows. Driver Signing Policy. `https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-`. Accessed: 2017-04-09 Last updated: 2016-09-08.

[53] Windows. Kernel Patch Protection. `https://msdn.microsoft.com/en-us/library/windows/hardware/Dn613955(v=vs.85).aspx`. Accessed: 2017-04-09 Last updated: 2007-01-22.

[54] W. Wong and M. Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, 2006.

[55] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

[56] C. Yang, R. C. Harkreader, and G. Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *International Workshop on Recent Advances in Intrusion Detection*, pages 318–337. Springer, 2011.

[57] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.

[58] A. Zaki and B. Humphrey. Unveiling the kernel: Rootkit discovery using selective automated kernel memory differencing. In *Proceedings of the 2014 VIRUS BULLETIN CONFERENCE*, 2014.

doi: 10.2313/NET-2017-09-1_11