# Malware and Attack Technologies Knowledge Area
# Version 1.0.1

**Wenke Lee** | Georgia Institute of Technology

**EDITOR**
**Howard Chivers** | University of York

**REVIEWERS**
**Alex Berry** | FireEye
**Lorenzo Cavallaro** | King's College London
**Mihai Christodorescu** | VISA
**Igor Muttik** | Cyber Curio

# COPYRIGHT



The CyBOK project would like to understand how the CyBOK is being used and its uptake. The project would like organisations using, or intending to use, CyBOK for the purposes of education, training, course development, professional development etc. to contact it at **contact@cybok.org** to let the project know how they are using CyBOK.

Version 1.0.1 is a stable public release of the Malware and Attack Technologies Knowledge Area.

# CHANGELOG

| Version date | Version number | Changes made |
|---|---|---|
| July 2021 | 1.0.1 | Updated copyright statement; amended “issue” to “version” |
| October 2019 | 1.0 | |

# INTRODUCTION

Malware is short for 'malicious software', that is, any program that performs malicious activities. We use the terms malware and malicious code interchangeably. Malware comes with a wide range of shapes and forms, and with different classifications accordingly, e.g., viruses, Trojans, worms, spyware, botnet malware, ransomware, etc.

Malware carries out many of the cyberattacks on the Internet, including nation-state cyberwar, cybercrime, fraud and scams. For example, Trojans can introduce a backdoor access to a government network to allow nation-state attackers to steal classified information. Ransomware can encrypt data on a user's computer and thus making it unaccessible to the user, and only decrypt the data after the user pays a sum of money. Botnet malware is responsible for many of the Distributed Denial-of-Service (DDoS) attacks as well as spam and phishing activities. We need to study the techniques behind malware development and deployment in order to better understand cyberattacks and develop the appropriate countermeasures.

As the political and financial stakes become higher, the sophistication and robustness of both the cyber defence mechanisms and the malware technologies and operation models have also increased. For example, attackers now use various obfuscation techniques such as packing and polymorphism as well as metamorphism to evade malware detection systems [1], and they set up adaptive network infrastructures on the Internet to support malware updates, command-and-control, and other logistics such as transits of stolen data. In short, it is becoming more important but also more challenging to study malware.

The rest of this chapter is organised as follows. We will provide a taxonomy of malware and discuss their typical malicious activities as well as their eco-system and support infrastructures. We will then describe the tools and techniques to analyse malware behaviours, and network- and host- based detection methods to identify malware activities, as well as processes and techniques including forensic analysis and attribution to respond to malware attacks.

# CONTENT

## 1 A TAXONOMY OF MALWARE

[2, c6]

There are many types of malware [2]. It is instructive to create a taxonomy to systematically categorise the wide spectrum of malware types. This taxonomy describes the common characteristics of each type of malware and thus can guide the development of countermeasures applicable to an entire category of malware (rather than a specific malware). Since there many facets of malware technologies and attack operations, based on which malware can be categorised and named, our taxonomy can include many dimensions. We discuss a few important ones below. It should be borne in mind that other, more specialised, attributes could also be used such as target processor architecture or operating system.

The first dimension of our taxonomy is whether malware is a standalone (or, independent) program or just a sequence of instructions to be embedded in another program. Standalone malware is a complete program that can run on its own once it is installed on a compromised machine and executed. For example, worms and botnet malware belong to this type. The second type requires a host program to run, that is, it must infect a program on a computer by inserting its instructions into the program so that when the program is run, the malware instructions are also executed. For example, document macro viruses and malicious browser plug-ins belong to this type. In general, it is easier to detect standalone malware because it is a program or a running process in its own right and its presence can be detected by operating system or security tools.

The second dimension is whether malware is persistent or transient. Most malware is installed in persistent storage (typically, a file system) as either standalone malware or an infection of another program that already resides in persistent storage. Other malware is memory-resident such that if the computer is rebooted or the infected running program terminates, it no longer exists anywhere on the system. Memory-resident malware can evade detection by many anti-virus systems that rely on file scanning. Such transient malware also has the advantage of being easy to clean up (or, cover-up) its attack operations. The traditional way for malware to become memory-resident is to remove the malware program (that was downloaded and installed previously) from the file system as soon as it gets executed. Newer approaches exploit system administrative and security tools such as PowerShell to inject malware directly into memory [3]. For example, according to one report [4], after an initial exploit that led to the unauthorised execution of PowerShell, meterpreter code was downloaded and injected into memory using PowerShell commands and it harvested passwords on the infected computer.

The third dimension generally applies to only persistent malware and categorises malware based on the layer of the system stack the malware is installed and run on. These layers, in the ascending order, include firmware, boot-sector, operating system kernel, drivers and Application Programing Interfaces (APIs), and user applications. Typically, malware in the lower layers is harder to detect and remove, and wreaks greater havoc because it has more control of the compromised computer. On the other hand, it is also harder to write malware that can be installed at a lower layer because there are greater constraints, e.g., a more limited programming environment in terms of both the types and amount of code allowed.

The fourth dimension is whether malware is run and spread automatically vs. activated by a user action. When an auto-spreading malware runs, it looks for other vulnerable machines on the Internet, compromises these machines and installs itself on them; the copies of malware on these newly infected machines immediately do the same – run and spread. Obviously, auto-spreading malware can spread on the Internet very quickly, often being able to exponentially increase the number of compromised computers. On the other hand, user-activated malware is run on a computer only because a user accidentally downloads and executes it, e.g., by clicking on an attachment or URL in a received email. More importantly, when this malware runs, although it can 'spread', e.g., by sending email with itself as the attachment to contacts in the user's address book, this spreading is not successful unless a user who receives this email activates the malware.

The fifth dimension is whether malware is static or one-time vs. dynamically updated. Most modern malware is supported by an infrastructure such that a compromised computer can receive a software update from a malware server, that is, a new version of the malware is installed on the compromised computer. From an attacker's point-of-view, there are many benefits of updating malware. For example, updated malware can evade detection techniques

that are based on the characteristics of older malware instances.

The sixth dimension is whether malware acts alone or is part of a coordinated network (i.e., a botnet). While botnets are responsible for many cyberattacks such as DDoS, spam, phishing, etc., isolated malware has become increasingly common in the forms of targeted attack. That is, malware can be specifically designed to infect a target organisation and perform malicious activities according to those assets of the organisation valuable to the attacker.

Most modern malware uses some form of obfuscation in order to avoid detection (and hence we do not explicitly include obfuscation in this taxonomy). There is a range of obfuscation techniques and there are tools freely available on the Internet for a malware author to use. For example, polymorphism can be used to defeat detection methods that are based on 'signatures' or patterns of malware code. That is, the identifiable malware features are changed to be unique to each instance of the malware. Therefore, malware instances look different from each other, but they all maintain the same malware functionality. Some common polymorphic malware techniques include packing, which involves compressing and encrypting part of the malware, and rewriting identifiable malicious instructions into other equivalent instructions.

| | standalone or host-program | persistent or transient | layers of system stack | auto-spreading? | dynamically updatable? | coordinated? |
|---|---|---|---|---|---|---|
| viruses | host-program | persistent | firmware and up | Y | Y | N |
| malicious browser extensions | host-program | persistent | application | N | Y | Y |
| botnet malware | both | persistent | kernel and up | Y | Y | Y |
| memory-resident malware | standalone | transient | kernel and up | Y | Y | Y |

Table 1: Use of the Taxonomy to Classify Representative Malware

As an illustration, we can apply this taxonomy to several types (or names) of malware. See Table 1. In particular, a virus needs a host-program to run because it infects the host-program by inserting a malicious code sequence into the program. When the host-program runs, the malicious code executes and, in addition to performing the intended malicious activities, it can look for other programs to infect. A virus is typically persistent and can reside in all layers of the system stack except hardware. It can spread on its own because it can inject itself into programs automatically. A virus can also be dynamically updated provided that it can connect to a malware update server. A polymorphic malware virus can mutate itself so that new copies look different, although the algorithm of this mutation is embedded into its own code. A virus is typically not part of a coordinated network because while the infection can affect many computers, the virus code typically does not perform coordinated activities.

Other malware that requires a host-program includes malicious browser plug-ins and extensions, scripts (e.g., JavaScript on a web page), and document macros (e.g., macro viruses and PDF malware). These types of malware can be updated dynamically, form a coordinated network, and can be obfuscated.

Botnet malware refers to any malware that is part of a coordinated network with a botnet infrastructure that provides command-and-control. A botnet infrastructure typically also

provides malware update, and other logistic support. Botnet malware is persistent and typically obfuscated, and usually resides in the kernel, driver, or application layers. Some botnet malware requires a host-program, e.g., malicious browser plug-ins and extensions, and needs user activation to spread (e.g., malicious JavaScript). Other botnet malware is standalone, and can spread automatically by exploiting vulnerable computers or users on the Internet. These include trojans, key-loggers, ransomware, click bots, spam bots, mobile malware, etc.

### 1.1 Potentially unwanted programs (PUPs)

A potentially unwanted program (PUP) is typically a piece of code that is part of a useful program downloaded by a user. For example, when a user downloads the free version of a mobile game app, it may include adware, a form of PUP that displays ad banners on the game window. Often, the adware also collects user data (such as geo-location, time spent on the game, friends, etc.) without the user's knowledge and consent, in order to serve more targeted ads to the user to improve the effectiveness of the advertising. In this case, the adware is also considered spyware, which is defined as unwanted program that steals information about a computer and its users. PUPs are in a grey area because, while the download agreement often contains information on these questionable behaviours, most users tend not to read the finer details and thus fail to understand exactly what they are downloading.

From the point of view of cybersecurity, it is prudent to classify PUPs towards malware, and this is the approach taken by many security products. The simple reason is that a PUP has all the potential to become full-fledged malware; once it is installed, the user is at the mercy of the PUP operator. For example, a spyware that is part of a spellchecker browser extension can gather information on which websites the user tends to visit. But it can also harvest user account information including logins and passwords. In this case, the spyware has become a malware from just a PUP.

## 2 MALICIOUS ACTIVITIES BY MALWARE

[2, c6][1, c11-12]

Malware essentially codifies the malicious activities intended by an attacker. Cyberattacks can be analysed using the Cyber Kill Chain Model [5], which, as shown in Table 2, represents (iterations of) steps typically involved in a cyberattack. The first step is Reconnaissance where an attacker identifies or attracts the potential targets. This can be accomplished, for example, by scanning the Internet for vulnerable computers (i.e., computers that run network services, such as sendmail, that have known vulnerabilities), or sending phishing emails to a group of users. The next phase is to gain access to the targets, for example, by sending crafted input to trigger a vulnerability such as a buffer overflow in the vulnerable network service program or embedding malware in a web page that will compromise a user's browser and gain control of his computer. This corresponds to the Weaponization and Delivery (of exploits) steps in the Cyber Kill Chain Model. Once the target is compromised, typically another piece of malware is downloaded and installed; this corresponds to the Installation (of malware) step in the Cyber Kill Chain Model. This latter malware is the real workhorse for the attacker and can carry out a wide range of activities, which amount to attacks on:

- confidentiality – it can steal valuable data, e.g., user's authentication information, and financial and health data;
- integrity – it can inject falsified information (e.g., send spam and phish emails, create fraudulent clicks, etc.) or modify data;
- availability – it can send traffic as part of a distributed denial-of-service (DDoS) attack, use up a large amount of compute-resources (e.g., to mine cryptocurrencies), or encrypt valuable data and demand a ransom payment.

| | Step | Activities |
|---|---|---|
| 1 | Reconnaissance | Harvesting email addresses, identifying vulnerable computers and accounts, etc. |
| 2 | Weaponization | Designing exploits into a deliverable payload. |
| 3 | Delivery | Delivering the exploit payload to a victim via email, Web download, etc. |
| 4 | Exploitation | Exploiting a vulnerability and executing malicious code on the victim's system. |
| 5 | Installation | Installing (additional) malware on the victim's system. |
| 6 | Command & Control | Establishing a command and control channel for attackers to remotely commandeer the victim's system. |
| 7 | Actions on Objectives | Carrying out malicious activities on the victim's system and network. |

Table 2: The Cyber Kill Chain Model

Most modern malware performs a combination of these attack actions because there are toolkits (e.g., a key-logger) freely available for carrying out many 'standard' activities (e.g., recording user passwords) [1], and malware can be dynamically updated to include or activate new activities and take part in a longer or larger 'campaign' rather than just performing isolated, one-off actions. These are the Actions on Objectives in the Cyber Kill Chain Model.

Botnets exemplify long-running and coordinated malware. A botnet is a network of bots (or, compromised computers) under the control of an attacker. Botnet malware runs on each bot and communicates with the botnet command-and-control (C&C) server regularly to receive instructions on specific malicious activities or updates to the malware. For example, every day the C&C server of a spamming botnet sends each bot a spam template and a list of email addresses so that collectively the botnet sends a very large number of spam messages. If the botnet is disrupted because of detection and response actions, e.g., the current C&C server is taken down, the botnet malware is already programmed to contact an alternative server and can receive updates to change to a botnet that uses peer-to-peer for C&C. In general, botnets are quite noisy, i.e., relatively easy to detect, because there are many bots in many networks. Botnet C&C is an example of the Command & Control step in the Cyber Kill Chain Model.

In contrast to botnets, malware behind the so-called advanced persistent threats (APTs) typically targets a specific organisation rather than aiming to launch large-scale attacks. For example, it may look for a particular type of controller in the organisation to infect and cause it to send the wrong control signals that lead to eventual failures in machineries. APT malware is typically designed to be long-lived (hence the term 'persistent'). This means it not only receives regular updates. but also evades detection by limiting its activity volume and intensity (i.e., 'low and slow'), moving around the organisation (i.e., 'lateral movements') and covering its tracks. For example, rather than sending the stolen data out to a 'drop site' all at once, it can send a small piece at a time and only when the server is already sending legitimate traffic; after it has finished stealing from a server it moves to another (e.g., by exploiting the trust

relations between the two) and removes logs and even patches the vulnerabilities in the first server.

When we use the Cyber Kill Chain Model to analyze a cyberattack, we need to examine its activities in each step. This requires knowledge of the attack techniques involved. The ATT&CK Knowledge Base [6] documents the up-to-date attack tactics and techniques based on real-world observations, and is a valuable reference for analysts.

### 2.1 The Underground Eco-System

The early-day malware activities were largely nuisance attacks (such as defacing or putting graffiti on an organisation's web page). Present-day malware attacks are becoming full-blown cyberwars (e.g., attacks on critical infrastructures) and sophisticated crimes (e.g., ransomware, fake-AntiVirus tools, etc.). An underground eco-system has also emerged to support the full malware lifecycle that includes development, deployment, operations and monetisation. In this eco-system, there are actors specialising in key parts of the malware lifecycle, and by providing their services to others they also get a share of the (financial) gains and rewards. Such specialisation improves the quality of malware. For example, an attacker can hire the best exploit researcher to write the part of the malware responsible for remotely compromising a vulnerable computer. Specialisation can also provide plausible deniability or at the least limit liability. For example, a spammer only 'rents' a botnet to send spam and is not guilty of compromising computers and turning them into bots; likewise, the exploit 'researcher' is just experimenting and not responsible for creating the botnet as long as he did not release the malware himself. That is, while they are all liable for the damage by malware, they each bear only a portion of the full responsibility.

## 3 MALWARE ANALYSIS

[1, c1-10] [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

There are many benefits in analysing malware. First, we can understand the intended malicious activities to be carried out by the malware. This will allow us to update our network and endpoint sensors to detect and block such activities, and identify which machines have the malware and take corrective actions such as removing it or even completely wiping the computer clean and reinstalling everything. Second, by analysing the malware structure (e.g., the libraries and toolkits that it includes) and coding styles, we may be able to gain information that is potentially useful to attribution, which means being able to identify the likely author and operator. Third, by comparing it with historical as well as geo-location data, we can better understand and predict the scope and trend of malware attacks, e.g., what kinds of activities (e.g., mining cryptocurrencies) are on the rise and if a cybercrime is moving from one region to another. In short, malware analysis is the basis for detecting and responding to cyberattacks.

Malware analysis typically involves running a malware instance in an analysis environment. There are ways to 'capture' malware instances on the infection sites. A network sensor can examine traffic (e.g., web traffic, email attachment) to identify possible malware (e.g., payload that contains binary or program-like data from a website with a low reputation) and run it in a sandbox to confirm. If a network sensor is able to detect outgoing malicious traffic from an internal host, a host-based sensor can further identify the program, i.e., the malware, responsible for such traffic. There are also malware collection and sharing efforts

where trusted organisations can upload malware samples found in their networks and also receive samples contributed by other organisations. Academic researchers can typically just obtain malware samples without needing to contribute. When acquiring and sharing malware samples, we must consider our legal and ethical responsibilities carefully [19]. For example, we must protect the identities of the infection sites from which the malware samples were captured, and we must not share the malware samples with any organisation that is an unknown entity or that does not have the commitment or technical capabilities to analyse malware safely.

The malware analysis pipeline typically includes the following steps: 1) identifying the format of a malware sample (e.g., binary or source code, Windows or Linux, etc.), 2) static analysis using disassembly (if the malware is in binary format), program analysis, statistical analysis of the file contents, etc., and 3) dynamic analysis using an analysis environment. Steps 2 and 3 can be combined and iterated.

## 3.1 Analysis Techniques

Malware analysis is the process of learning malware behaviours. Due to the large volume and increasing complexity of malware, we need to be able to rapidly analyse samples in a complete, reliable and scalable way. To achieve this, we need to employ techniques such as static analysis, dynamic analysis, symbolic execution and concolic execution [1]. These program analysis techniques have been developed to support the software development cycle, and they often need to be customized or extended for malware analysis because malicious programs typically include code constructed specifically to resist analysis. That is, the main challenge in malware analysis is to detect and bypass anti-analysis mechanisms.

### 3.1.1 Static Analysis

Static analysis involves examining the code (source, intermediate, or binary) to assess the behaviours of a program without actually executing it [1]. A wide range of malware analysis techniques fall into the category of static analysis. One limitation is that the analysis output may not be consistent with the actual malware behaviours (at runtime). This is because in many cases it is not possible to precisely determine a program's behaviours statically (i.e., without the actual run-time input data). A more serious problem is that malware authors are well aware of the limitations of static analysis and they leverage code obfuscation and packing to thwart static-analysis altogether. For example, the packed code cannot be statically analysed because it is encrypted and compressed data until unpacked into executable code at run-time.

### 3.1.2 Dynamic analysis

Dynamic analysis monitors the behaviours of malware execution in order to identify malicious behaviours [1]. Static analysis can provide more comprehensive coverage of program behaviours but may include unfeasible ones. Dynamic analysis identifies the precise program behaviours per the test input cases but misses behaviours that are not triggered by the input. Additionally, dynamical analysis can defeat code obfuscation techniques designed to evade static analysis. For example, when malware at run-time unpacks and executes its packed code, dynamic analysis is able to identify the (run-time) malicious behaviours in the originally packed code. When performing dynamic analysis, the main questions to consider are: what types of malicious behaviours need to be identified and correspondingly, what run-time features need to be collected and when to collect (or sample), and how to isolate the effects on the malware from those of benign system components. Typically, the run-time features to be collected need to be from a layer lower than the malware itself in the system stack so that the malware cannot change the collected information. For example, instruction traces certainly cover all the details of malicious behaviours but the data volume is too large for efficient analysis [20]. On the other hand, system call (or API call) traces are coarser but summarise how malware interacts with the run-time system, including file I/O and networking activities [21]. Another advantage of dynamic analysis is that it is independent of the malware format, e.g., binary, script, macro, or exploit, because all malware is executed and analysed in a similar fashion.

### 3.1.3 Fuzzing

Fuzzing is a method for discovering vulnerabilities, bugs and crashes in software by feeding randomised inputs to programs. Fuzzing tools [22] can also be used to trigger malware behaviours. Fuzzing can explore the input space, but it is limited due to code-coverage issues [7], especially for inputs that drive the program down complex branch conditions. In contrast, concolic execution (see 3.1.5 Concolic Execution) is good at finding complex inputs by formulating constraints, but is also expensive and slow. To take advantage of both approaches, a hybrid approach [23] called *hybrid fuzzing* can be used.

### 3.1.4 Symbolic Execution

Symbolic execution [24, 25, 26, 7, 10] has been used for vulnerability analysis of legitimate programs as well as malware analysis [8]. It treats variables and equations as symbols and formulas that can potentially express all possible program paths. A limitation of concrete execution (i.e., testing on particular inputs), including fuzzing, for malware analysis is that the program has to be executed end-to-end, one run at a time. Unlike concrete execution, symbolic execution can explore multiple branches simultaneously. To explore unseen code sections and unfold behaviours, symbolic execution generalises the input space to represent all possible inputs that could lead to points of interest.

### 3.1.5 Concolic Execution

While symbolic execution can traverse all paths in theory, it has major limitations [24], e.g., it may not converge quickly (if at all) when dealing with large symbol space and complex formulas and predicates. Concolic execution, which combines *CONC*rete and symb*OLIC* execution, can reduce the symbolic space but keep the general input space.

*Offline Concolic Execution* is a technique that uses concrete traces to drive symbolic execution; it is also known as a *Trace Based Executor* [9]. The execution trace obtained by concrete execution is used to generate the path formulas and constraints. The path formulas for the corresponding branch is negated and Satisfiability Modulo Theories (SMT) solvers are used to find a valid input that can satisfy the not-taken branches. Generated inputs are fed into the program and re-run from the beginning. This technique iteratively explores the feasible not-taken branches encountered during executions. It requires the repetitive execution of all the instructions from the beginning and knowledge of the input format.

*Online Concolic Execution* is a technique that generates constraints along with the concrete execution [10]. Whenever the concrete execution hits a branch, if both directions are feasible, execution is forked to work on both branches. Unlike the offline executor, this approach can explore multiple paths.

*Hybrid Execution:* This approach switches automatically between online and offline modes to avoid the drawbacks of non-hybrid approaches [11].

Concolic Execution can use whole-system emulators [10, 27] or dynamic binary instrumentation tools [11, 25]. Another approach is to interpret Intermediate Representation (IR) to imitate the effects of execution [8, 12]. This technique allows context-free concolic execution, which analyses any part of the binary at function and basic block levels.

*Path Exploration* is a systematical approach to examine program paths. Path explosion is also inevitable in concolic execution due to the nature of symbolic space. There are a variety of algorithms used to prioritise the directions of concolic execution, e.g., Depth-First Search (DFS) or distance computation [28]. Another approach is to prioritise the directions favouring newly explored code blocks or symbolic memory dependence [11]. Other popular techniques include path pruning, state merging [10, 29, 30], under-constrained symbolic execution [12] and fuzzing support [7, 9].

## 3.2 Analysis Environments

Malware analysis typically requires a dedicated environment to run the dynamic analysis tools [1]. The design choice of the environment determines the analysis methods that can be utilised and, therefore, the results and limitations of analysis. Creating an environment requires balancing the cost it takes to analyse a malware sample against the richness of the resulting report. In this context, cost is commonly measured in terms of time and manual human effort. For example, having an expert human analyst study a sample manually can produce a very in-depth and thorough report, but at great cost. Safety is a critical design consideration because of the concern that malware being executed and analysed in the environment can break out of its containment and cause damage to the analysis system and its connected network including the Internet (see 3.2.1 Safety and Live-Environment Requirements). An example is running a sample of a botnet malware that performs a DDoS attack, and thus if the analysis environment is not safe, it will contribute to that attack.

| | Machine Emulator | Type 2 Hypervisor | Type 1 Hypervisor | Bare-metal machine |
|---|---|---|---|---|
| **Architecture** | Code-based architecture emulation | Runs in host OS, provides virtualisation service for hardware | Runs directly on system hardware | No virtualisation |
| **Advantages** | Easy to use, Fine-grained introspection, Powerful control over the system state | Easy to use, Fine-grained introspection, Powerful control over the system state | Medium transparency, Fine-grained introspection, Low overhead for hardware interaction | High transparency, No virtual environment artifacts |
| **Disadvantages** | Low transparency, Unreliability support of architecture semantics | Low transparency, Artifacts from para-virtualisation | Less control over the system state | Lack of fine-grained introspection, Scalability and cost issues, Slower to restore to clean state |
| **Examples** | Unicorn [31], QEMU [32], Bochs [33] | VirtualBox [34], KVM [35], VMware [36] | VMwareESX [37], Hyper-V [38], Xen [39] | NVMTrace [40], BareCloud [16] |

Table 3: Comparison of Malware Analysis Environments

Table 3 highlights the advantages and disadvantages of common environments used for run-time (i.e., dynamic) analysis of malware. We can see that some architectures are easier to set up and give finer control over the malware's execution, but come at the cost of transparency (that is, they are easier for the malware to detect) compared to the others. For example, bare-metal systems are very hard for malware to detect, but because they have no instrumentation, the data that can be extracted are typically limited to network and disk I/O. By contrast, emulators like QEMU can record every executed instruction and freely inspect memory. However, QEMU also has errors that do not exist in real hardware, which can be exploited to detect its presence [41]. A very large percentage of modern malware detect emulated and virtualised environments and if they do, then they do not perform their malicious actions in order to avoid analysis.

### 3.2.1 Safety and Live-Environment Requirements

Clearly, *safety* is very important when designing a malware analysis environment because we cannot allow malware to cause unintended damage to the Internet (e.g., via mounting a denial-of-service attack from inside the analysis environment) and the analysis system and its connected network. Unfortunately, although pure static techniques, i.e., code analysis without program execution, are the safest, they also have severe limitations. In particular, malware authors know their code may be captured and analysed, and they employ code obfuscation techniques so that code analysis alone (i.e., without actually running the malware) will yield as little information as possible.

Malware typically requires communication with one or more C&C servers on the Internet, e.g., to receive commands and decrypt and execute its 'payload' (or the code that performs the intended malicious activities). This is just one example that highlights how the design of a live-environment is important for the malware to be *alive* and thus exhibit its intended functionality. Other examples of live-environment requirements include specific run-time

libraries [42], real user activities on the infected machine [43], and network connectivity to malware update servers [44].

### 3.2.2 Virtualised Network Environments

Given the safety and live-environment requirements, most malware analysis environments are constructed using virtualisation technologies. Virtualisation enables operating systems to automatically and efficiently manage entire networks of nodes (e.g., hosts, switches), even within a single physical machine. In addition, containment policies can be applied on top of the virtual environments to balance the live-environment and safety requirements to 1) allow malware to interact with the Internet to provide the necessary realism, and 2) contain any malicious activities that would cause undesired harm or side-effects.

Example architectures [13] include: 1) the GQ system, which is designed based on multiple containment servers and a central gateway that connects them with the Internet allowing for filtering or redirection of the network traffic on a per-flow basis, and 2) the Potemkin system, which is a prototype *honeyfarm* that uses aggressive memory sharing and dynamically binds physical resources to external requests. Such architectures are used to not only monitor, but also replay network-level behaviours. Towards this end, we first need to reverse-engineer the C&C protocol used by malware. There are several approaches based on network level data (e.g., Roleplay [45], which uses bytestream alignment algorithms), or dynamic analysis of malware execution (e.g., Polyglot and dispatcher [46]), or a combination of the two.

## 3.3 Anti-Analysis and Evasion Techniques

Malware authors are well aware that security analysts use program analysis to identify malware behaviours. As a result, malware authors employ several techniques to make malware hard to analyse [1].

### 3.3.1 Evading the Analysis Methods

The source code of malware is often not available and, therefore, the first step of static analysis is to disassemble malware binary into assembly code. Malware authors can apply a range of anti-disassembly techniques (e.g., reusing a byte) to cause disassembly analysis tools to produce an incorrect code listing [1].

The most general and commonly used code obfuscation technique is *packing*, that is, compressing and encrypting part of the malware. Some trivially packed binaries can be unpacked with simple tools and analysed statically [47], but for most modern malware the packed code is unpacked only when it is needed during malware execution. Therefore, an unpacking tool needs to analyse malware execution and consider the trade-offs of robustness, performance, and transparency. For example, unpackers based on virtual machine introspection (VMI) [14] are more transparent and robust but also slower. By contrast, unpackers built on dynamic binary instrumentation (DBI) [18] are faster, but also easier to detect because the DBI code runs at the same privilege level as the malware.

Many techniques aim at obfuscating the intended control-flows of a malware, e.g., by adding more basic blocks and edges to its control-flow graph [1, 48, 49]. A countermeasure is to analyze malware samples by their dynamic features (i.e., what a malware does). The reason is that static analysis can be made impossible via advanced obfuscation using opaque

constants [50], which allows the attacker to hide what values will be loaded into registers during runtime. This in turn makes it very hard for static malware analysis to extract the control-flow graph and variables from the binary. A more effective approach is to combine static and dynamic analysis. For example, such an approach has been shown to be able to disassemble the highly obfuscated binary code [51].

A less common but much more potent obfuscation technique is code emulation. Borrowing techniques originally designed to provide software copyright protection [52], malware authors convert native malware binaries into bytecode programs using a randomly generated instruction set, paired with a native binary emulator that interprets the instruction set. That is, with this approach, the malware 'binary' is the emulator, and the original malware code becomes 'data' used by the emulator program. Note that, for the same original malware, the malware author can turn it into many instances of emulated malware instances, each with its own random bytecode instruction set and a corresponding emulator binary. It is extremely hard to analyse emulated malware. Firstly, static analysis of the emulator code yields no information about the specific malware behaviours because the emulator processes all possible programs in the bytecode instruction set. Static analysis of the malware bytecode entails first understanding the instruction set format (e.g., by static analysing the emulator first), and developing tools for the instruction set; but this process needs to be repeated for every instance of emulated malware. Secondly, standard dynamic analysis is not directly useful because it observes the run-time instructions and behaviours of an emulator and not of the malware.

A specialised dynamic analysis approach is needed to analyse emulated malware [17]. The main idea is to execute the malware emulator and record the entire instruction traces. Applying dynamic dataflow and taint analysis techniques to these traces, we then identify data regions containing the bytecode, syntactic information showing how bytecodes are parsed into opcodes and operands, and semantic information about control transfer instructions. The output of this approach is data structures, such as a control-flow graph (CFG) of the malware, which provides the foundation for subsequent malware analysis.

Malware often uses fingerprinting techniques to detect the presence of an analysis environment and evade dynamic analysis (e.g., it stops executing the intended malware code). More generally, malware behaviours can be 'trigger-based' where a trigger is a run-time condition that must be true. Examples of conditions include the correct date and time, the presence of certain files or directories, an established connection to the Internet, the absence of a specific mutex object etc. If a condition is not true, the malware does not execute the intended malicious logic. When using standard dynamic analysis, the test inputs are not guaranteed to trigger some of these conditions and, as a result, the corresponding malware behaviours may be missed. To uncover trigger-based behaviours a multi-path analysis approach [15] explores multiple execution paths of a malware. The analyser monitors how the malware code uses condition-like inputs to make control-flow decisions. For each decision point, the analyser makes a snapshot of the current malware execution state and allows the malware to execute the correct malware path for the given input value; for example, the input value suggests that the triggering condition is not met and the malware path does not include the intended malicious logic. The analyser then comes back to the snapshot and rewrites the input value so that the other branch is taken; for example, now the triggering condition is rewritten to be true, and the malware branch is the intended malicious logic.

### 3.3.2 Identifying the Analysis Environments

Malware often uses system and network artifacts that suggest that it is running in an analysis environment rather than a real, infected system [1]. These artifacts are primarily categorised into four classes: virtualisation, environment, process introspection, and user. In virtualisation fingerprinting, evasive malware tries to detect that it is running in a virtualised environment. For example, it can use red pill testing [53], which entails executing specific CPU instruction sequences that cause overhead, unique timing skews, and discrepancies when compared with executions on a bare-metal (i.e., non-virtualised) system. Regarding environment artifacts, virtual machines and emulators have unique hardware and software parameters including device models, registry values, and processes. In process introspection, malware can check for the presence of specific programs on operating systems, including monitoring tools provided by anti-virus companies and virtual machine vendors. Lastly, user artifacts include specific applications such a web browser (or lack thereof), web browsing history, recently used files, interactive user prompts, mouse and keyboard activities etc. These are signals for whether a real human uses the environment for meaningful tasks.

An analysis environment is not transparent if it can be detected by malware. There are mitigation techniques, some address specific types of evasion while others more broadly increase transparency. Binary modifications can be performed by dynamically removing or rewriting instructions to prevent detection [54], and environmental artifacts can be hidden from malware by hooking operating system functions [55]. Path-exploration approaches [15, 56] force malware execution down multiple conditional branches to bypass evasion. Hypervisor-based approaches [14, 57] use introspection tools with greater privilege than malware so that they can be hidden from malware and provide the expected answers to the malware when it checks the system and network artifacts. In order to provide the greatest level of transparency, several approaches [40, 16] perform malware analysis on real machines to avoid introducing artifacts.

# 4 MALWARE DETECTION

[1, c11, c14-16, c18] [58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68]

## 4.1 Identifying the Presence of Malware

The process of locating a malicious program residing within a host involves identifying clues that are indicative of the malware's presence on a computer system. We call these clues 'indicator of compromise', and they are the 'features' or 'artifacts' of malware.

### 4.1.1 Finding Malware in a Haystack

In order to identify malware, we must first have an understanding of how malware is distributed to their victims' hosts. Malware is commonly distributed via an Internet download [69]. A vulnerable Internet-facing program running on a computer can be exploited to download malware onto the computer. A user on the computer can be socially engineered to open an email attachment or visit a web page, both may lead to an exploit and malware download.

Whilst being downloaded onto a host, the malware's contents can be seen in the payload section of the network traffic (i.e., network packet) [1]. As a defense, an Antivirus (AV) solution, or Intrusion Detection System (IDS), can analyse each network packet transported to an end-host for known malicious content, and block (prevent) the download. On the other hand, traffic content encrypted as HTTPS is widely and increasingly adopted by websites. Using domain reputation systems [70], network traffic coming from domains and IP addresses known to be associated with malicious activities can be automatically blocked without analysing the traffic's payload.

After being installed on a computer, malware can reside within the host's filesystem or memory (or both). At this point, the malware can sleep (where the executable does nothing to the system) until a later point in time [71] as specified by the malware author. An AV or IDS can periodically scan the host's filesystem and memory for known malicious programs [1]. As a first layer of defence, malware detectors can analyse static features that suggest malicious executable contents. These include characteristics of instructions, control-flow graphs, call graphs, byte-value patterns [72] etc.

If malware is not detected during its distribution state, i.e., a detection system misses its presence in the payloads of network traffic or the filesystem and memory of the end-host, it can still be detected when it executes and, for example, begins contacting its command-and-control (C&C) server and performing malicious actions over the Internet or on the victim computer system. An AV or IDS on the network perimeter continuously monitors network packets travelling out of an end-host. If the AV or IDS sees that the host is contacting known malicious domain names or IP addresses it can surmise that the host has been infected by malware. In addition, an AV or IDS on the end-host can look for behaviour patterns that are associated with known malware activities, such as system or API calls that reveal the specific files read or written.

**Evasion and Countermeasures** Since Antivirus and IDS solutions can generate signatures for malware executables, malware authors often morph the contents of their malware. They can change the contents of the executables while generating identically functional copies of their malware (i.e., the malware will perform the same dynamic behaviours when executed). Since its static contents have been changed, the malware can evade an AV or IDS that uses these static features. On the other hand, the malware can still be detected by an AV or IDS that uses the dynamic features (i.e., what the malware does).

Heuristics, e.g., signatures of a packing tool, or high entropy due to encryption, can be used to detect and block contents that suggest the presence of packed malware, but this may lead to false alarms because packing can also be used by benign software and services, such as video games, to protect proprietary information. The most reliable way to detect packed malware is to simply monitor its run-time behaviours because the packed code will be unpacked and executed, and the corresponding malicious behaviours can then be identified [58].

In addition to changing the malware executable, an attacker can also change the contents

of its malicious network traffic by using *polymorphism* to modify payloads so that the same attacks look different across multiple traffic captures. However, classic polymorphic malware techniques [73] make the payloads look so different that even a naive IDS can easily differentiate them from benign payloads. On the other hand, with polymorphic malware blending attacks [59] malicious payloads can be made to look statistically similar to benign payloads.

Malware authors often implement updating routines, similar to updates for operating systems and applications such as web browsers and office tools. This allows malware authors the flexibility to make changes to the malware to not only include new malicious activities but also evade detection by AVs and IDS that have started using patterns of the old malware and its old behaviours.

## 4.2 Detection of Malware Attacks

We have discussed ways to identify static and behaviour patterns of malware, which can then be used to detect instances of the same, or similar malware. Although many popular variants of malware families have existed at one time or another (e.g., Zeus [74, 75], Spyeye [76, 77], Mirai [78]), there will always be new malware families that cannot be detected by malware detection models (such as AV signatures). Therefore, we need to go beyond identifying specific malware instances: we need to detect malicious activities in general.

### 4.2.1 Host-based and Network-Based Monitoring

The most general approach to detect malicious activities is anomaly detection [60, 79, 61]. An anomaly in system or network behaviour is an activity that deviates from normal (or seen) behaviour. Anomaly detection can identify both old and new attacks. It is important to note that an *anomalous* behaviour is not the same as a *malicious* behaviour. Anomalous behaviours describe behaviours that deviate from the norm, and of course it is possible to have abnormal benign activities occurring on a system or network.

On the other hand, a more efficient and arguably more accurate approach to detect an old attack is to find the patterns or signatures of the known attack activities [1]. This is often called the misuse detection approach. Examples of signatures include: unauthorised write to system files (e.g., Windows Registry), connection to known botnet C&C servers, etc.

Two different, but complementary approaches to deploy attack detection systems are: 1) host-based monitoring of system activities, and 2) network-based monitoring of traffic. Host-based monitoring systems monitor activities that take place in a host, to determine if the host is compromised. These systems typically collect and monitor activities related to the file system, processes, and system calls [1, 62]. Network-based monitoring systems analyse activities that are network-wide, e.g., temporal characteristics of access patterns of network traffic flows, the domain names the network hosts reach out to, the characteristics of the network packet payloads that cross the network perimeter, etc. [1, 63].

Let us look at several examples of malicious activities and the corresponding detection approaches. The first-generation spam detection systems focused on analysing the email contents to distinguish legitimate messages from spam. Latter systems included network-level behaviours indicative of spam traffic [80], e.g., spikes in email traffic volumes due to large amount of spam messages being sent.

For DDoS detection, the main idea is to analyse the statistical properties of traffic, e.g., the

number of requests within a short time window sent to a network server. Once a host is identified to be sending such traffic, it is considered to be participating in a DDoS attack and its traffic is blocked. Attackers have evolved their techniques to DDoS attacks, in particular, by employing multiple compromised hosts, or bots, to send traffic in a synchronised manner, e.g., by using DDoS-as-a-service malware kits [81]. That is, each bot no longer needs to send a large amount of traffic. Correspondingly, DDoS detection involves correlating hosts that send very similar traffic to the victim at the same time.

For ransomware detection, the main approaches include monitoring host activities involved in encryption. If there is a process making a large number of *significant* modifications to a large number of files, this is indicative of a ransomware attack [82]. The '*significant*' modifications reflect the fact that encrypting a file will result in its contents changing drastically from its original contents.

Host-based and network-based monitoring approaches can be beneficially combined. For example, if we see contents from various sensitive files on our system (e.g., financial records, password-related files, etc.) being transmitted in network traffic, it is indicative that data are being exfiltrated (without the knowledge and consent of the user) to an attacker's server. We can then apply host-based analysis tools to further determine the attack provenance and effects on a victim host [83].

Since many malicious activities are carried out by botnets, it is important to include botnet detection methods. By definition, bots of the same botnet are controlled by the same attacker and perform coordinated malicious activities [84, 64]. Therefore, a general approach to botnet detection is to look for synchronised activities both in C&C like traffic and malicious traffic (e.g., scan, spam, DDoS, etc.) across the hosts of a network.

### 4.2.2 Machine Learning-Based Security Analytics

Since the late 1990s, machine learning (ML) has been applied to automate the process of building models for detecting malware and attacks. The benefit of machine learning is its ability to generalise over a population of samples, given various features (descriptions) of those samples. For example, after providing an ML algorithm samples of different malware families for 'training', the resultant model is able to classify new, unseen malware as belonging to one of those families [65].

Both static and dynamic features of malware and attacks can be employed by ML-based detection models. Examples of static features include: instructions, control-flow graphs, call graphs, etc. Examples of dynamic features include: system call sequences and other statistics (e.g., frequency and existence of system calls), system call parameters, data-flow graphs [85], network payload features, etc.

An example of success stories in applying machine learning to detect malware and attacks is botnet detection [86]. ML techniques were developed to efficiently classify domain names as ones produced by domain generation algorithm (DGA), C&C domains, or legitimate domains using features extracted from DNS traffic. ML techniques have also been developed to identify C&C servers as well as bots in an enterprise network based on features derived from network traffic data [64].

A major obstacle in applying (classical) machine learning to security is that we must select or even engineer features that are useful in classifying benign and malicious activities. Feature engineering is very knowledge- and labour- intensive and is the bottleneck in applying ML to

any problem domain. Deep learning has shown some promise in learning from a large amount of data without much feature engineering, and already has great success in applications such as image classification [87]. However, unlike many classical ML models (such as decision trees and inductive rules) that are human-readable, and hence reviewable by security analysts before making deployment decisions, deep learning outputs blackbox models that are not readable and not easily explainable. It is often not possible to understand what features are being used (and how) to arrive at a classification decision. That is, with deep learning, security analysts can no longer check if the output even makes sense from the point-of-view of domain or expert knowledge.

### 4.2.3 Evasion, Countermeasures, and Limitations

Attackers are well aware of the detection methods that have been developed, and they are employing evasion techniques to make their attacks hard to detect. For example, they can limit the volume and intensity of attack activities to stay below the detection threshold, and they can mimic legitimate user behaviours such as sending stolen data (a small amount at a time) to a 'drop site' only when a user is also browsing the Internet. Every misuse or anomaly detection model is potentially evadable.

It should also come as no surprise that no sooner had researchers begun using ML than attackers started to find ways to defeat the ML-based detection models.

One of the most famous attacks is the Mimicry attack on detection models based on system call data [66]. The idea is simple: the goal is to morph malicious features to look exactly the same as the benign features, so that the detection models will mistakenly classify the attack as benign. The Mimicry attack inserts system calls that are inconsequential to the intended malicious actions so that the resultant sequences, while containing system calls for malicious activities, are still legitimate because such sequences exist in benign programs. A related attack is polymorphic blending [59] that can be used to evade ML models based on network payload statistics (e.g., the frequency distribution of n-grams in payload data to a network service). An attack payload can be encoded and padded with additional n-grams so that it matches the statistics of benign payloads. Targeted noise injection [67] is an attack designed to trick a machine-learning algorithm, while training a detection model, to focus on features not belonging to malicious activities at all. This attack exploits a fundamental weakness of machine learning: garbage in, garbage out. That is, if you give a machine-learning algorithm bad data, then it will learn to classify data 'badly'. For example, an attacker can insert various no-op features into the attack payload data, which will statistically produce a strong signal for the ML algorithm to select them as 'the important, distinguishing features'. As long as such features exist, and as they are under the attacker's control, any ML algorithm can be misled to learn an incorrect detection model. Noise injection is also known as 'data poisoning' in the machine learning community.

We can make attacks on ML harder to succeed. For example, one approach is to squeeze features [88] so that the feature set is not as obvious to an attacker, and the attacker has a smaller target to hit when creating adversarial samples. Another approach is to train separating classes, which distance the decision boundary between classes [89]. This makes it more difficult for an attacker to simply make small changes to features to 'jump' across decision boundaries and cause the model to misclassify the sample. Another interesting approach is to have an ML model forget samples it has learned over time, so that an attacker has to continuously poison every dataset [90].

A more general approach is to employ a combination of different ML-based detection models so that defeating all of them simultaneously is very challenging. For example, we can model multiple feature sets simultaneously through ensemble learning, i.e., using multiple classifiers trained on different feature sets to classify a sample rather than relying on singular classifier and feature set. This would force an attacker to have to create attacks that can evade each and every classifier and feature set [68].

As discussed earlier, deep learning algorithms produce models that cannot be easily examined. But if we do not understand how a detection model really works, we cannot foresee how attackers can attempt to defeat it and how we can improve its robustness. That is, a model that seemingly performs very well on data seen thus far can, in fact, be very easily defeated in the future - we just have no way of knowing. For example, in image recognition it turned out that some deep learning models focused on high-frequency image signals (that are not visible to the human eye) rather than the structural and contextual information of an image (which is more relevant for identifying an object) and, as a result, a small change in the high-frequency data is sufficient to cause a mis-classification by these models, while to the human eye the image has not changed at all [91].

There are promising approaches to improve the 'explainability' of deep learning models. For example, an attention model [92] can highlight locations within an image to show which portions it is focusing on when classifying the image. Another example is LEMNA [93], which generates a small set of interpretable features from an input sample to explain how the sample is classified, essentially approximating a local area of the complex deep learning decision boundary using a simpler interpretable model.

In both the machine learning and security communities, adversarial machine learning [94] is and will continue to be a very important and active research area. In general, attacks on machine learning can be categorised as data poisoning (i.e., injecting malicious noise into training data) and evasion (i.e., morphing the input to cause mis-classification). What we have discussed above are just examples of evasion and poisoning attacks on ML models for security analytics. These attacks have motivated the development of new machine-learning paradigms that are more robust against adversarial manipulations, and we have discussed here examples of promising approaches.

In general, attack detection is a very challenging problem. A misuse detection method which is based on patterns of known attacks is usually not effective against new attacks or even new variants of old attacks. An anomaly detection method which is based on a normal profile can produce many false alarms because it is often impossible to include all legitimate behaviours in a normal profile. While machine learning can be used to automatically produce detection models, potential 'concept drift' can render the detection models less effective over time [95]. That is, most machine-learning algorithms assume that the training data and the testing data have the same statistical properties, whereas in reality, user behaviours and network and system configurations can change after a detection model is deployed.

# 5 MALWARE RESPONSE

[96, 97, 98, 99, 100, 101]

If we have an infected host in front of us, we can remove the malware, and recover the data and services from secure backups. At the local network access point, we can update corresponding Firewall and Network intrusion detection system rules, to prevent and detect future attacks. It is unfeasible to execute these remediation strategies if the infected machines cannot be accessed directly (e.g., they are in private residences), and if the scale of infection is large. In these cases, we can attempt to take down malware command-and-control (C&C) infrastructure instead [96, 97], typically at the internet service provider (ISP) or the top-level domain (TLD) level. Takedowns aim to disrupt the malware communication channel, even if the hosts remain infected. Last but not least, we can perform attack attribution using multiple sources of data to identify the actors behind the attack.

## 5.1 Disruption of Malware Operations

There are several types of takedowns to disrupt malware operations. If the malware uses domain names to look up and to communicate with centralised C&C servers, we perform takedown of C&C domains by 'sinkholing' the domains, i.e., making the C&C domains resolve to the defender's servers so that botnet traffic is 'trapped' (that is, redirected) to these servers [96]. If the malware uses peer-to-peer (P2P) protocol as a decentralised C&C mechanism, we can partition the P2P botnet into isolated sub-networks, create a sinkholing node, or poison the communication channel by issuing commands to stop the malicious activities [97]. However, it should be borne in mind that, in most territories active defence or intelligence gathering, such as hack-backs, access to or modification of servers, DNS, or networks, is unlawful without appropriate legal authority.

### 5.1.1 Evasion and Countermeasures

Malware often utilises agility provided by DNS fast-flux network and Domain-name Generation Algorithms (DGAs) to evade the takedown. A DNS fast-flux network points the C&C domain names to a large pool of compromised machines, and the resolution changes rapidly [102]. DGAs make use of an algorithm to automatically generate candidate C&C domains, usually based on some random seed. Among the algorithm-generated domains, the botmaster can pick a few to register (e.g., on a daily basis) and make them resolve to the C&C servers. What makes the matter worse are the so-called bullet-proof hosting (BPH) services, which are resilient against takedowns because they ignore abuse complaints and takedown requests [98].

We can detect the agile usage of C&C mechanisms. As the botmaster has little control of the IP address diversity and down-time for compromised machines in a fast-flux network, we can use these features to detect fast-flux [103]. We can also identify DGA domains by mining NXDomains traffic using infected hosts features and domain name characteristic features [86], or reverse-engineering the malware to recover the algorithm. To counter bullet-proof hosting, we need to put legal, political and economic pressures on hosting providers. For example, the FBI's operation ghost click issued a court order for the takedown of DNSChanger [104, 105].

Malware has also become increasingly resilient by including contingency plans. A centralised botnet can have P2P as a fallback mechanism in case the DNS C&C fails. Likewise, a P2P botnet can use DNS C&C as a contingency plan. A takedown is effective only if all the C&C

channels are removed from the malware. Otherwise, the malware can bootstrap the C&C communication again using the remaining channels. If we hastily conduct botnet takedowns without thoroughly enumerating and verifying all the possible C&C channels, we can fail to actually disrupt the malware operations and risk collateral damage to benign machines. For example, the Kelihos takedown [106] did not account for the backup P2P channel, and the 3322.org takedown disabled the dynamic DNS service for many benign users.

We need to have a complete view of the C&C domains and other channels that are likely to be used by a botnet, by using multiple sources of intelligence including domain reputation, malware query association and malware interrogation [96]. We start from a seed set of C&C domains used by a botnet. Then, we use passive DNS data to retrieve related historical IP addresses associated with the seed set. We remove sinkholing, parking, and cloud hosting provider IP addresses from them to mitigate the collateral damage from the takedowns. The resulting IPs can also give us related historical domains that have resolved to them. After following these steps, we have an extended set of domains that are likely to be used by the botnet. This set captures agile and evasive C&C behaviours such as fast-flux networks. Within the extended set, we combine 1) low reputation domains, 2) domains related to malware, and 3) other domains obtained by interrogating the related malware. Malware interrogation simulates situations where the default C&C communication mechanism fails through blocking DNS resolution and TCP connection [101]. By doing so, we can force the malware to reveal the backup C&C plans, e.g., DGA or P2P. After enumerating the C&C infrastructure, we can disable the complete list of domains to take the botnet down.

## 5.2 Attribution

Ideally, law enforcement wants to identify the actual criminal behind the attacks. Identifying the virtual attacker is an important first step toward this goal. An attacker may have consistent coding styles, reuse the same resources or infrastructures, or use similar C&C practices.

From the malware data, we can compare its 'characteristics' with those of known historical adversaries, e.g., coding styles, server configurations, etc. [99]. At the source code level, we can use features that reflect programming styles and code quality. For instance, linguistic features, formatting style, bugs and vulnerabilities, structured features such as execution path, abstract syntax tree (AST), Control Flow Graph (CFG), and program dependence graph (PDG) can be used. Other features extracted from the binary file can also indicate authorship, e.g., the sequence of instructions and register flow graph.

From the enumerated attack infrastructure, we can associate the expanded domain name set with previously known adversaries. For instance, unknown TDSS/TDL4 botnet ad-fraud C&C domains share the same IP infrastructure with known domains, and they are registered by the same set of email addresses and name servers. This allows us to attribute unknown domains to known TDSS/TDL4 actors [100].

### 5.2.1 Evasion and Countermeasures

Many malware authors reuse different kits for the convenience offered by the business model of the underground economy. Common for-sale kits allow malware authors to easily customise their own malware. They can also evade attribution by intentionally planting 'false flags' in malware.

Domain registration information, WHOIS, is a strong signal for attack attribution. The same attacker often uses a fake name, address and company information following a pattern. However, WHOIS privacy protection has become ubiquitous and is even offered for free for the first year when a user purchases a domain name. This removes the registration information that could be used for attack attribution.

We need to combine multiple, different streams of data for the analysis. For instance, malware interrogation helps recover more C&C domains used by the fallback mechanism, which offers more opportunity for attribution [101, 107].

## CONCLUSION

Attackers use malware to carry out malicious activities on their behalf. Malware can reside in any layer of the system stack, and can be a program by itself or embedded in another application or document. Modern malware comes with a support infrastructure for coordinated attacks and automated updates, and can operate low-and-slow and cover its tracks to avoid detection and attribution. While malware can cause wide-spread infection and harm on the Internet, it can also be customised for attacks targeting a specific organisation. Malware analysis is an important step in understanding malicious behaviours and properly updating our attack prevention and detection systems. Malware employs a wide range of evasion techniques, which include detecting the analysis environment, obfuscating malicious code, using trigger-conditions to execute, and applying polymorphism to attack payloads, etc. Accordingly, we need to make analysis environments transparent to malware, continue to develop specialised program analysis algorithms and machine-learning based detection techniques, and apply a combination of these approaches. Response to malware attacks goes beyond detection and mitigation, and can include take-down and attribution, but the challenge is enumerating the entire malware infrastructure, and correlating multiple pieces of evidence to avoid false flags planted by the attackers.

## CROSS-REFERENCE OF TOPICS VS REFERENCE MATERIAL

| Sections | Cites |
|---|---|
| 1 A taxonomy of Malware | [2]:c6 |
| 2 Malicious Activities by Malware | [2]:c6, [1]:c11-12 |
| 3 Malware Analysis | |
| 3.1 Analysis Techniques | [1]:c1-10 |
| 3.1.1 Static Analysis | [1]:c4-7 |
| 3.1.2 Dynamic analysis | [1]:c8-10 |
| 3.1.3 Fuzzing | [7, 8] |
| 3.1.5 Concolic Execution | [9, 10, 11, 12] |
| 3.2 Analysis Environments | [1]:c2 |
| 3.2.1 Safety and Live-Environment Requirements | |
| 3.2.2 Virtualised Network Environments | [1]:c2, [13] |
| 3.3.2 Identifying the Analysis Environments | [1]:c15-18, [14, 15, 16] |
| 3.3 Anti-Analysis and Evasion Techniques | [1]:c15-16, [17, 18, 15] |
| 4 Malware Detection | |
| 4.1 Identifying the Presence of Malware | |
| 4.1.1 Finding Malware in a Haystack | [1]:c11,c14 |
| 4.1.1 Evasion and Countermeasures | [1]:c15-16,c18, [58, 59] |
| 4.2 Detection of Malware Attacks | |
| 4.2.1 Host-based and Network-Based Monitoring | [1]:c11,c14, [60, 61, 62, 63, 64] |
| 4.2.2 Machine Learning-Based Security Analytics | [65, 64] |
| 4.2.3 Evasion, Countermeasures, and Limitations | [66, 67, 68] |
| 5 Malware Response | |
| 5.1 Disruption of Malware Operations | [96, 97] |
| 5.1.1 Evasion and Countermeasures | [98] |
| 5.2 Attribution | [99, 100] |
| 5.2.1 Evasion and Countermeasures | [101] |

# REFERENCES

[1] M. Sikorski and A. Honig, *Practical Malware Analysis: A Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[2] W. Stallings and L. Brown, *Computer Security: Principles and Practice, 4th Edition*. Pearson, 2018.

[3] McAfee, "Fileless malware execution with powershell is easier than you may realize," 2017. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf

[4] ars TECHNICA, "A rash of invisible, fileless malware is infecting banks around the globe," 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/02/a-rash-of-invisible-fileless-malware-is-infecting-banks-around-the-globe/?comments=1&post=32786675

[5] Lockheed Martin, "The cyber kill chain." [Online]. Available: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html

[6] MITRE, "ATT&CK knowledge base." [Online]. Available: https://attack.mitre.org

[7] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok: state of the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *The*

*Network and Distributed System Security Symposium (NDSS)*, 2008.
[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2012, pp. 380–394.
[12] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code." in *USENIX Security Symposium*, 2015, pp. 49–64.
[13] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson, "GQ: Practical containment for measuring modern malware systems," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 2011, pp. 397–412.
[14] 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*. ACM, 2008, pp. 51–62.
[15] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *IEEE Symposium on Security and Privacy*. IEEE, 2007.
[16] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection." in *USENIX Security Symposium*, 2014, pp. 287–301.
[17] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 94–109.
[18] S. Mariani, L. Fontana, F. Gritti, and S. D'Alessio, "PinDemonium: a DBI-based generic unpacker for Windows executables," in *Black Hat USA 2016*, 2016.
[19] E. Kenneally, M. Bailey, and D. Maughan, "A framework for understanding and applying ethical principles in network and security research," in *Workshop on Ethics in Computer Security Research (WECSR '10)*, 2010.
[20] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and API call sequences," *Journal in computer virology*, vol. 7, no. 2, pp. 107–119, 2011.
[21] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 178–197.
[22] M. Zalewski, "American fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/
[23] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium*, 2018.
[24] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," in *Communications of the ACM*, 2013.
[25] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011.
[26] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.
[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*. Springer, 2008, pp. 1–25.
[28] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2329–2344.
[29] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic

execution," *ACM Sigplan Notices*, vol. 47, no. 6, pp. 193–204, 2012.
[30] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1083–1094.
[31] "The unicorn emulator." [Online]. Available: https://www.unicorn-engine.org/
[32] "The QEMU emulator." [Online]. Available: https://www.qemu.org/
[33] "The bochs emulator." [Online]. Available: http://bochs.sourceforge.net/
[34] "The VirtualBox." [Online]. Available: https://www.virtualbox.org/
[35] "The KVM." [Online]. Available: https://www.linux-kvm.org/
[36] "The VMware." [Online]. Available: https://www.vmware.com/
[37] "The VMware ESXi." [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html/
[38] "The Hyper-V." [Online]. Available: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/
[39] "The Xen." [Online]. Available: https://www.xenproject.org/
[40] P. Royal, "Entrapment: Tricking malware with transparent, scalable malware analysis," 2012, talk at Black Hat.
[41] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *International Conference on Information Security*. Springer, 2007, pp. 1–18.
[42] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, "Understanding Linux Malware," in *IEEE Symposium on Security & Privacy*, 2018.
[43] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1009–1024.
[44] J. T. Bennett, N. Moran, and N. Villeneuve, "Poison ivy: Assessing damage and extracting intelligence," *FireEye Threat Research Blog*, 2013.
[45] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog." in *NDSS*, 2006.
[46] J. Caballero and D. Song, "Automatic protocol reverse-engineering: Message format extraction and field semantics inference," *Computer Networks*, vol. 57, no. 2, pp. 451–474, 2013.
[47] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," in *European Symposium on Research in Computer Security*. Springer, 2008, pp. 481–500.
[48] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 290–299.
[49] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *NDSS*, 2008.
[50] 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*. IEEE, 2007, pp. 421–430.
[51] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 745–756.
[52] "Vmprotect." [Online]. Available: https://vmpsoft.com
[53] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and AntiVM technologies," in *Anti-Disassembly*

*and Anti-VM Technologies, Black Hat USA Conference*, 2012.
[54] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *IEEE Symposium on Security and Privacy*. IEEE, 2006.
[55] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security & Privacy*, vol. 5, no. 2, 2007.
[56] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-executing binary programs for security applications," in *The 23rd USENIX Security Symposium (USENIX Security 14*, 2014, pp. 829–844.
[57] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," *ACM Sigplan Notices*, vol. 47, no. 7, pp. 227–238, 2012.
[58] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 289–300.
[59] P. Fogla, M. I. Sharif, R. Perdisci, O. M. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *USENIX Security*, 2006.
[60] D. Denning and P. G. Neumann, *Requirements and model for IDES-a real-time intrusion-detection expert system*. SRI International, 1985.
[61] H. S. Javitz and A. Valdes, "The NIDES statistical component: Description and justification," *Contract*, vol. 39, no. 92-C, p. 0015, 1993. [Online]. Available: http://www.csl.sri.com/papers/statreport/
[62] K. Ilgun, R. Kemmerer, and P. Porras, "State transition analysis: A rule-based intrusion detection approach," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 181–199, 1995.
[63] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.
[64] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th USENIX Security Symposium (Security'08)*, 2008.
[65] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*. IEEE, 1999, pp. 120–132.
[66] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM, 2002, pp. 255–264.
[67] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.
[68] A. Kantchelian, J. D. Tygar, and A. D. Joseph, "Evasion and hardening of tree ensemble classifiers," *arXiv preprint arXiv:1509.07892*, 2015.
[69] G. Cleary, M. Corpin, O. Cox, H. Lau, B. Nahorney, D. O'Brien, B. O'Gorman, J.-P. Power, S. Wallace, P. Wood, and C. Wueest, "Internet security threat report," Symantec, Tech. Rep., 2018.
[70] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster, "Building a dynamic reputation system for DNS," in *USENIX security symposium*, 2010, pp. 273–290.
[71] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 285–296.
[72] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in

*International Workshop on Recent Advances in Intrusion Detection*. Springer, 2004, pp. 203–222.

[73] P. Szor, *The Art of Computer Virus Research and Defense*. Symantec Press, 2005, ch. Advanced code evolution techniques and computer virus generator kits.

[74] K. Stevens and D. Jackson, "Zeus banking trojan report," *Atlanta: SecureWorks*, 2010.

[75] N. Falliere and E. Chien, "Zeus: King of the bots," Symantec, Tech. Rep. Security Response, 2009. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/security-response-zeus-king-of-bots-09-en.pdf

[76] B. Krebs, "Feds to charge alleged SpyEye trojan author." [Online]. Available: https://krebsonsecurity.com/2014/01/feds-to-charge-alleged-spyeye-trojan-author/#more-24554

[77] D. Gilbert, "Inside SpyEye: How the russian hacker behind the billion-dollar malware was taken down," Oct 2017, international Business Times. [Online]. Available: https://www.ibtimes.com/inside-spyeye-how-russian-hacker-behind-billion-dollar-malware-was-taken-down-2357477

[78] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the Mirai botnet," in *USENIX Security Symposium*, 2017, pp. 1093–1110.

[79] 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*. IEEE, 1996, pp. 120–128.

[80] S. Hao, A. Kantchelian, B. Miller, V. Paxson, and N. Feamster, "Predator: proactive recognition and elimination of domain abuse at time-of-registration," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1568–1579.

[81] C. Rossow, "Amplification hell: Revisiting network protocols for DDoS abuse." in *NDSS*, 2014.

[82] D. Y. Huang, D. McCoy, M. M. Aliapoulios, V. G. Li, L. Invernizzi, E. Bursztein, K. McRoberts, J. Levin, K. Levchenko, and A. C. Snoeren, "Tracking ransomware end-to-end," in *Tracking Ransomware End-to-end*. IEEE Symposium on Security & Privacy, 2018.

[83] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *27th USENIX Security Symposium*. USENIX Association, 2018, pp. 1705–1722.

[84] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting malware infection through IDS-driven dialog correlation," in *Proceedings of the 16th USENIX Security Symposium (Security'07)*, August 2007.

[85] 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*, 2009, pp. 351–366.

[86] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: Detecting the rise of DGA-based malware," in *USENIX security symposium*, vol. 12, 2012.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[88] W. Xu, D. Evans, and Y. Qi, "Feature squeezing: Detecting adversarial examples in deep neural networks," *arXiv preprint arXiv:1704.01155*, 2017.

[89] M. McCoyd and D. Wagner, "Background class defense against adversarial examples," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 96–102.

[90] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 463–480.
[91] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 39–57.
[92] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
[93] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS '18)*, 2018.
[94] N. Dalvi, P. Domingos, S. Sanghai, D. Verma, and others, "Adversarial classification," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 99–108.
[95] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *Proceedings of the 26th USENIX Security Symposium*, 2017.
[96] Y. Nadji, M. Antonakakis, R. Perdisci, D. Dagon, and W. Lee, "Beheading hydras: Performing effective botnet takedowns," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. ACM, 2013, pp. 121–132.
[97] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "SoK: P2PWNED-modeling and evaluating the resilience of peer-to-peer botnets," in *IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 97–111.
[98] S. Alrwais, X. Liao, X. Mi, P. Wang, X. Wang, F. Qian, R. Beyah, and D. McCoy, "Under the shadow of sunshine: Understanding and detecting bulletproof hosting on legitimate service provider networks," in *IEEE Symposium on Security and Privacy*. IEEE, 2017, pp. 805–823.
[99] S. Alrabaee, P. Shirani, M. Debbabi, and L. Wang, "On the feasibility of malware authorship attribution," in *International Symposium on Foundations and Practice of Security*. Springer, 2016, pp. 256–272.
[100] Y. Chen, P. Kintis, M. Antonakakis, Y. Nadji, D. Dagon, W. Lee, and M. Farrell, "Financial lower bounds of online advertising abuse," in *International conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 231–254.
[101] Y. Nadji, M. Antonakakis, R. Perdisci, and W. Lee, "Understanding the prevalence and use of alternative plans in malware with network games," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 1–10.
[102] M. Konte, N. Feamster, and J. Jung, "Fast flux service networks: Dynamics and roles in hosting online scams," Georgia Institute of Technology, Tech. Rep., 2008.
[103] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and detecting fast-flux service networks." in *NDSS*, 2008.
[104] FBI New York Field Office, "Operation ghost click: International cyber ring that infected millions of computers dismantled," April 2012. [Online]. Available: https://www.fbi.gov/news/stories/international-cyber-ring-that-infected-millions-of-computers-dismantled
[105] W. Meng, R. Duan, and W. Lee, "DNS changer remediation study," in *M3AAWG 27th General Meeting*, 2013.
[106] *Civil Action No: 1:11cv1017 (JCC/IDD), Microsoft Corporation v. Dominique Alexander Piatti, Dotfree Group SRO John Does 1–22, Controlling a computer botnet thereby injuring Microsoft and its customers*. UNITED STATES DISTRICT COURT FOR THE EASTERN DISTRICT OF VIRGINIA, Feb 2013.
[107] B. Bartholomew and J. A. Guerrero-Saade, "Wave your false flags!" [Online]. Available:

https://securelist.com/wave-your-false-flags/76273/

## ACRONYMS

**API** Application Programing Interface.

**APT** Advanced Persistent Threat.

**AST** Abstract Syntax Tree.

**AV** AntiVirus.

**BPH** Bullet Proof Hosting.

**C&C** Command and Control.

**CAPTCHA** Completely Automated Public Turing test to tell Computers and Humans Apart.

**CFG** Control Flow Graph.

**CPU** Central Processing Unit.

**DBI** Dynamic Binary Instrumentation.

**DDoS** Distributed Denial of Service.

**DFS** Depth-First Search.

**DGA** Domain-name Generation Algorithm.

**DNS** Domain Name System.

**IDS** Intrusion Detection System.

**IR** Intermediate Representation.

**ISP** Internet Service Provider.

**ML** Machine Learning.

**OS** Operating System.

**P2P** Peer to Peer.

**PDG** Program Dependence Graph.

**PUP** Potentially Unwanted Program.

**SMT** Satisfiability Modulo Theories.

**SWIFT** Society for Worldwide Interbank Financial Telecommunication.

**TCP** Transmission Control Protocol.

**TLD** Top Level Domain.

**URL** Uniform Resource Locator.

**VMI** Virtual Machine Inspection.

## GLOSSARY

**advanced persistent threat** An attack to an organization that continues its activities and yet remains undetected for an extended period of time.

**botnet** A network of compromised computers (or, bots) that is controlled by an attacker to launch coordinated malicious activities.

**CyBOK** Refers to the Cyber Security Body of Knowledge.

**exploit** Software or data that takes advantage of a vulnerability in a system to cause unintended consequences. (Source = NCSC Glossary).

**indicator of compromise** Recognised action, specific, generalized, or theoretical, that an adversary might be expected to take in preparation for an attack. (Source = NIST IR 7298).

**key-logger** A virus or physical device that logs keystrokes to secretly capture private information such as passwords or credit card details.(Source = BSI Glossary).

**macro virus** A virus that attaches itself to documents and uses the macro programming capabilities of the document's application to execute and propagate.(Source = NIST IR 7298).

**malware** A program inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications or operating system, or of otherwise annoying or disrupting the victim. Synonym = malicious code. (Source = NIST IR 7298r2).

**malware analysis** The process of analyzing malware code and understanding its intended functionalities.

**malware detection** The process of detecting the presence of malware in a system.

**metamorphic malware** Malware of which each iteration or instance has different code from the preceding one. The code changes make it difficult to recognize the different iterations are the same malware (contrast with polymorphic malware).

**meterpreter** A tool that allows an attacker to control a victim's computer by running an invisible shell and establishing a communication channel back to the attacking machine.

**packed malware** Packed malware is obfuscated malware in which the malicious program is compressed and cannot be analysed statically.

**packing** A technique to obfuscate malware (see packed malware).

**passive dns** A mechanism to collect large amounts of DNS data by storing DNS responses from servers. (Source = RFC7719.

**polymorphic malware** Malware that changes each instance to avoid detection. It typically has two parts: the decryptor and the encrypted program body. Each instance can encrypt the malware program differently and hence has a different decryptor; however, once decrypted, the same malware code is executed. (contrast with metamorphic malware).

**polymorphism** See polymorphic malware.

**potentially unwanted program** A program that may not be wanted by a user and is often downloaded along with a program that the user wants. Examples include adware, spyware, etc.

**ransomware** Malicious software that makes data or systems unusable until the victim makes a payment. (Source = NIST IR 7298).

**safety** In the context of malware analysis, a requirement that malware should be prevented from causing damage to the connected systems and networks while it runs in the analysis environment.

**sinkholing** A technique used by a DNS server to give out false information to prevent the use of a domain name.

**spam** The abuse of electronic messaging systems to indiscriminately send unsolicited bulk messages. (Source = NIST IR 7298).

**spyware** Software that is secretly or surreptitiously installed into an information system to gather information on individuals or organizations without their knowledge; a type of malicious code. (Source = NIST IR 7298).

**trojan** A computer program that appears to have a useful function, but also has a hidden and potentially malicious function that evades security mechanisms, sometimes by exploiting legitimate authorizations of a system entity that invokes the program. (Source = NIST IR 7298).

**virus** A hidden, self-replicating section of computer software, usually malicious logic, that propagates by infecting - i.e., inserting a copy of itself into and becoming part of - another program. A virus cannot run by itself; it requires that its host program be run to make the virus active. (Source = SANS security glossary).

**worm** A computer program that can run independently, can propagate a complete working version of itself onto other hosts on a network, and may consume computer resources destructively. (Source = SANS security glossary).

# INDEX