Check for updates

JDMS

Special Issue Article

Journal of Defense Modeling and Simulation: Applications, Methodology, Technology
2018, Vol. 15(1) 31–48
© The Author(s) 2017
DOI: 10.1177/1548512917721755
journals.sagepub.com/home/dms

Malware modeling and experimentation through parameterized behavior

Z. Berkay Celik[1], Patrick McDaniel[1], and Thomas Bowen[2]

Abstract

Experimentation is critical to understanding the malware operation and to evaluating potential defenses. However, constructing the controlled environments needed for this experimentation is both time-consuming and error-prone. In this study, we highlight several common mistakes made by researchers and conclude that existing evaluations of malware detection techniques often lack in both *flexibility* and *transparency*. For instance, we show that small variations in the malware's behavioral parameters can have a significant impact on the evaluation results. These variations, if unexplored, may lead to overly optimistic conclusions and detection systems that are ineffective in practice. To overcome these issues, we propose a framework to model malware behavior and guide systematic parameter selection. We evaluate our framework using a synthetic botnet executed within the CyberVAN testbed. Our study is intended to foster critical evaluation of proposed detection techniques and stymie unintentionally erroneous experimentation.

Keywords

Experimentation, malware modeling, simulation

1 Introduction

Malware experimentation is essential for researchers to understand the malware operation and develop potential defenses. To build effective defenses, detection systems need to be evaluated through understanding the reactions to parametrized real-world malware behaviors. However, there are few publicly available malware-execution datasets. Researchers operate malware samples in a controlled environment and observe the malware behavior.[1–4] Although such an operation holds significance for the community, constructing the controlled environments and running the malware is an error-prone endeavor. Such errors in experimental setup and malware implementation can, and do, propagate to results and subsequent analyses.

In this paper, we build a malware experimentation process and evaluate it with an execution of synthetic botnet in a controlled testbed. The framework includes a sequential model for implementation of the malware and a validation procedure. Malware implementation covers a concrete botnet example with parameters governing its behavior, and validation procedure that aims at finding the impact of the parameters on an evaluation of detection techniques. In this manner, we evaluate systems through comprehensive simulation of the malware with a parameterized behavior.

We use a process to identify several common mistakes researchers make and quantify the impact of those missteps. We demonstrate that even a single parameter setting in a malware configuration file changes the malware behavior. The unnatural timing heterogeneity between legitimate and malware traces example shows how defensive mechanisms may lead to optimistic conclusions – results in very few or even zero false positives. For the remediation of such fallacies, we present systematic parameter selection to foster critical evaluation of proposed detection

[1]SIIS Laboratory, Department of CSE, The Pennsylvania State University
[2]Vencore Labs, Basking Ridge, NJ, USA

Corresponding author:
Z. Berkay Celik, SIIS Laboratory, Department of CSE The Pennsylvania State University, University Park, PA 16802, USA.
Email: zbc102@cse.psu.edu

techniques and prevent erroneous experimentation. Summarizing, we make the following contributions:

- We propose an experimentation framework for operating the malware in a controlled environment that aims at a prudent evaluation.
- The framework covers implementation and execution of a synthetic botnet to model parameterized malware behavior.
- We propose a sensitivity analysis based validation procedure. The procedure aims at avoiding optimistic conclusions caused by unsystematic parameter configuration.
- We present recommendations on the operation, and experimentation of the malware that can be used by the security community to foster their experiments.

2 Background

In general, malware is concerned with two phases: (1) infection and (2) operation. In the infection phase, an attacker runs a small snippet of code on the victim's host for instance via a malicious website. The exploit is successful when a victim fetches and executes the code. In the operation phase, the malware program starts running, and it generates various malicious activities. These phases mainly cover the behavior of the malware through the configuration file, propagation techniques, and attack strategy.

Researchers examine these phases by use of the real-world traces in-the-wild or execute malware in a closed or controlled environment, and collect system or network level traces for assessing the proposed solutions. Real-world malware traces are either limited due to the privacy issues, being out of date, or have constraints on extending new attacks to the existing malware variants. These limitations lead researchers to obtain malware traces from a controlled environment. For example, some researchers attempt to solve the problem by obtaining the malware traces from honeynet systems,[5] others gather malware binary samples from repositories[6] and execute in a controlled lab environment.

For instance, BotMiner system executed two Internet Relay Chat (IRC) and Hypertext Transfer Protocol (HTTP) bot code (e.g., Spybot and Sdbot) in a controlled virtual network.[3] Livedas and Strayer et al. used a testbed within the BBN company's production network to obtain the botnet traces.[1,2] Similarly, Garcia et al. and Wurzinger et al. executed and monitored the infected machines with various malware families in a controlled laboratory.[4,7] Further, some researchers use testbeds such as Emulab,[8] PlanetLab,[9] and DETER[10] to execute and observe the malware characteristics in a closed environment.[11,12]

Although execution of malware using the ready-to-run binaries appears to be straightforward, observing its behavior under various parameters is a challenging task. Some malware may use a simple configuration file, and this does mutate its operation over time. For instance, recent malware variants dynamically change their port numbers for filtering data, define a period of heartbeat message for alive messages, and use different network protocols for propagation.[13,14] Such a file is a common practice among the real-world captured malware to obfuscate its activity and bypass filtering restrictions of the detection systems. Therefore, it is necessary to generate a comprehensive set of parameters in a controlled environment similar to those found in-the-wild.

Our analysis of recently published existing research results on controlled malware executions shows that their evaluations lack modeling the parameterized malware behavior.[1–4,11,15] Our analysis leads to the following observations: (a) the parameters that govern the malware's behavior is not often transparent, (b) malware binaries are executed using the constant parameter values, (c) despite the implementation differences of the malware, the detection rate is very high (very few or even zero false positives), and (d) the datasets used for the experimental evaluations are not often publicly available. The above observations create a gap of an appropriate level of detail to replicate and validate the evaluation results.

To address these problems, we draw inspiration from our experience in malware detection techniques.[16–20] We first present the requirements for an experimentation framework for prudent evaluation. Using this, we implement and operate a botnet in a controlled environment with a set of parameters that are used similarly to the parameters of the in the wild malware to model its behavior. To demonstrate and avoid the potential experimentation and parameter variability fallacies, we implement a validation procedure for operating the malware. We show examples of the parameters impacting the operation of a botnet binary through various parameter configurations. Finally, we present the details of the botnet implementation and share the experimental and operational efforts to help researchers to foster their evaluation of experimentation.

3 Experimental design requirements

We aim for malware experimentation in a controlled environment with specific capabilities to be operated. To do so, we investigate the recent malware reverse engineering research efforts to cover the malware characteristics.[13,21–26] These efforts establish requirements for an experimentation design that are critical to operating and exploring the parameterized malware behavior. Without such requirements, one may resort to a simple and

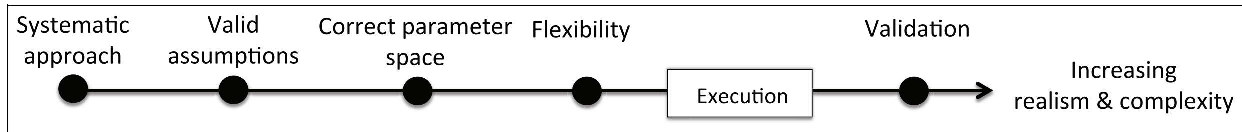

Figure 1. Experimentation design requirements: Comprehensive goals are necessary to implement and operate realistic malware in a controlled environment.

imprecise experimentation. We summarize this process in Figure 1 and discuss the details below.

1. **Systematic approach**– This represents the clear purpose of the problem to be solved. Definition of the problem helps to identify the correct adversarial assumptions and malware parameters influencing the overall implementation goals.
2. **Valid assumptions**– Evaluating a system's robustness in realistic, adversarial, and complex scenarios requires explicitly listing the assumptions. Such scenarios should match the environment through motivated and skilled attackers. Otherwise, simplifying assumptions need to be considered carefully, as imprecise assumptions about an attacker and malware capabilities may undermine the relevance.
3. **Correct parameter space**– This includes the malware execution under a various set of parameters, because these parameters may impact the malware behavior. The parameters include but are not limited to the number of infected hosts, propagation patterns, and communication protocols. The flexible setting of these parameter values and repeating the experiments renders the conclusions valid, as each of the parameters impacts the outcome of the malware behavior and provides a fine granularity of observations under different malware infections.
4. **Flexibility**– This refers to the ability for the implementation to adapt possible development, testing changes in the rapid evolution of real-world threats. The key is building a highly flexible malware design that is a loose coupling of malware lifecycle. Generally speaking, it includes developing a solution that is not tightly integrated with each component of infection, communication protocol, rally mechanism, evasion techniques, and malicious activities. The flexibility makes it easier for subsequent versions of the malware to update its code, adapting techniques and creating new variants.
5. **Validation**– The parameters and assumptions of any implementation are subject to change and error. Validation procedure is required for a reality check and reliable statistics report, as it often serves to challenge the implicit assumptions and the parameter space. These are needed to present the impacts on the conclusions to be drawn.

4 Botnet model and simulation

We present the details of the botnet design and its implementation in the Cyber Virtual Assured Network (CyberVAN) testbed. Figure 2 illustrates an overview of the five components of the malware design and a validation procedure. In this section, we present the design and implementation details of the each component. We provide the major engineering obstacles that we overcame to design the botnet. The implementation is made available on the project website (https://cybervan.appcomsci.com:9000/welcome). In Section 5, we run the botnet in the CyberVAN testbed and collect data to validate its behavior through a wide range of parameters governing its behavior.

Testbed Environment – We implement and run the malware in CyberVAN testbed.[27] The testbed is designed to enable functional testing and performance evaluation of distributed applications for supporting cyber security experimentation. It models hosts using various virtualization mechanisms and networks using discrete event network simulation. It supports a hybrid emulation environment with hosts running applications that exchange data over simulated wired and wireless networks. In contrast to other existing testbed facilities such as GENI,[28] DETER,[10] and National Cyber Range,[29] CyberVAN provides a flexible experimentation environment both for public and military networks. Further, it can be configurable to create a connection in the closed network nodes, and nodes in the testbed can connect other nodes that are out of the simulation environment through Internet IP path. A complete description and features of the CyberVAN are given in Chadha et al.[30]

Botnet Implementation – We present an implementation of a concrete botnet in which details are considered from the reverse engineering of malware found in-the-wild. The implementation includes the infection, operation, and propagation of a botnet developed to run in the CyberVAN testbed. The implementation is flexible; it

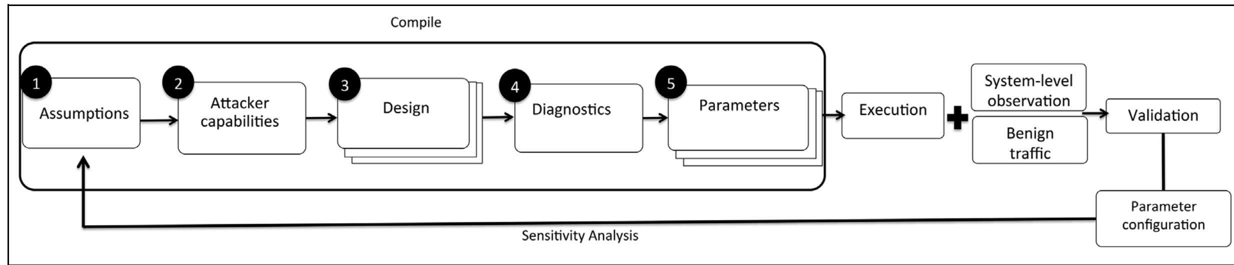

Figure 2. Overview of malware experimentation process: The process includes implementation and operation of the malware with various parameter configurations to explore the malware's behavioral variations, and a validation procedure to address the erroneous experimentation.

allows both executing arbitrary commands on the infected botnet victims and integrating additional vulnerabilities.

The implementation includes five components of a botnet architecture (see Figure 2). Each component is shown in rectangles, with directed edges between them indicating dependencies. The entire botnet design works sequentially, and each component is dependent on the one preceding it. (❶, ❷), these components generalize assumptions and attacker capabilities to form a sound foundation for the implementation process to proceed. (❸), a design component covers the elements of the infection, rendezvous point discovery, propagation and communication patterns. (❹), a component that captures various testbed and malware configuration logs as well as the errors for diagnosis process, and (❺), a component includes static and dynamic parameters to observe the malware's behavioral variations. We next detail these components.

4.1 Terminology

This section presents the definitions used in the subsequent sections.

- Attacker: Human operating the botnet to perform malicious activities.
- BotMaster: Software residing on the BotMaster computer to provide an interface for the botnet.
- BotMaster computer: Computer used by an attacker to run the BotMaster software.
- Victim: Computers under attacker's control.
- Botnet victim: Computers infected with the BotClient malware.
- Private name server computers: Computers on which attacker has the ability to configure and operate a lightweight name server application(e.g., Dnsmasq).
- Rendezvous point: Computers on which an attacker uses to configure and run the rendezvous service.
- BotClient: Malicious software residing on the victim botnet member computers.
- Public name server: The underlying legitimate name service infrastructure for resolving the host name lookups.
- Rendezvous service: It is a software residing on a rendezvous point computer that provides a bi-directional relay service between BotCients and BotMaster. Rendezvous service is an indirection intended to enhance stealth and robustness. Rendezvous service is not strictly required, as it does not change the message flow. However, BotClients can, if desired, communicate directly with rendezvous service to alter the flow of the network traffic.
- Control: Computer used to control the testbed scenario. The control computer has a SSH access to the testbed computers. In this way, the access to the host is preserved even if there is a connection failure among the hosts.

4.2 Assumptions

This section presents the assumptions of the botnet implementation as follows:

- Attacker has the ability to change name server entries for a few specific Domain Name System (DNS) zones.
- At least one botnet victim has occasional network connectivity to a rendezvous point.
- The rendezvous point have network connectivity to BotMaster.
- Initial infection of the first victim is through social engineering (e.g., an unsuspecting user is tricked into downloading and executing the BotClient).
- Subsequent infection of additional victim computers is through the exploitation of vulnerabilities on those computers and results in downloading and executing the BotClient. We consider the following BotClients that can be exploited to download files and execute arbitrary commands:

- *P2paa daemon.* P2paa is a daemon written explicitly to support the attack propagation. The p2paa configuration file (known as peers) on each computer identifies the p2paa peers of that computer and enables identification of other botnet victims.
- *SQL injection.* The SQL database executing on selected victims is vulnerable to SQL injection attacks.
- *Shell shock.* The shell executing on selected victims is vulnerable to the shell shock attack.

- Once a victim is infected, BotClients are able to self-propagate to other victims. Propagation requires reconnaissance to discover potential victims. Two forms of reconnaissance are implemented:

- *Stubbed.* The set of victims and their vulnerabilities are listed in header files compiled into BotClient and used to direct propagation from each victim. It corresponds to a real attack in which the attacker has previously mapped the victim network and has tailored the attack for that specific network.
- *Automated.* Victims actively scan for the new victims using a network mapping tool nmap upon infection.

4.3 Capabilities of the attacker

We present the capabilities of an attacker through a BotMaster as follows:

- `Execute()`: It specifies an arbitrary system command, which BotClients on all botnet victims execute. The results of the execution (`stdout and stderr`) are streamed back to the BotMaster for display.
- `Download()`: It specifies a source file with the full path name of a file existing on the attacker's computer and a destination file. The source file is downloaded to all victims and stored in the destination file.

The adjective system in execute command indicates that the command is a binary executable file. For instance, it is suitable as a parameter to the `execve` system call. We note that all of the standard Linux commands (e.g., `ls, tail, cat`, etc.) are virtually executable binaries, thus the limitation is minor. However, some shell scripts command cannot be executed as they would be from the command line. Rather, these commands must be executed as arguments to their interpreter such as in bash. The ability to download and execute a file gives the attacker virtually unlimited access to the victims. For instance, a simple exfiltration program that examines the victim file system for confidential text files and sends them back to the attacker is provided. The attacker can simply download and execute this file to perform exfiltration. BotMaster creates a separate `Xterminal` for each victim and routes output among the windows for the victims involved with the execution of a command.

The execute interface precludes the efficient use of interactive commands (e.g., vi). BotMaster has no capability to send additional input to an interactive command after the command has started. Further, the `stdout` and `stderr` streaming facility for returning results is unsuitable for any commands that require terminal rendering capabilities. We also collect periodic report status from BotClients and display in the Xterminal.

4.4 Design

The bulk of BotMaster, BotClient, and rendezvous service designs are trivial. It deals with mundane tasks such as collecting input from an attacker, using socket-based communication to move messages between the entities, and internal logic to move messages within the entities. However, four design objects are worth detailed treatment: (1) rendezvous point agreement between BotClients and BotMaster, (2) infection of BotClients to additional victims through vulnerabilities, (3) communication between entities, and (4) persistence of BotClients. We show the design objects in Figure 3 and discuss them in the following subsections.

4.4.1 Rendezvous point discovery. As botnet victims are infected with BotClient, they attempt to communicate with BotMaster through a rendezvous point. BotMaster periodically moves the rendezvous point among a set of the available rendezvous point for stealth. Therefore, the communication requires that BotMaster and BotClients come to a dynamic agreement upon the current rendezvous point. The agreement is reached by one of three different methods. It is determined by BotMaster and given as an argument before running the infected hosts as follows:

- Single Fast flux – BotMaster starts rendezvous service on one of the rendezvous points and registers it under a hard-coded name (notionally `RPNAME`) with the public name server. BotClients query public name server using `RPNAME`. If found, it uses the resolved address as a rendezvous point.
- Double Fast flux – BotMaster starts rendezvous service on one of the rendezvous points. Then, it starts a private name server on one of the name servers by registering `RPNAME` with the chosen rendezvous point on the private name server. BotMaster registers the selected private name server under a hard-coded name (notionally `NSNAME`) with the public name server. BotClient queries public name server

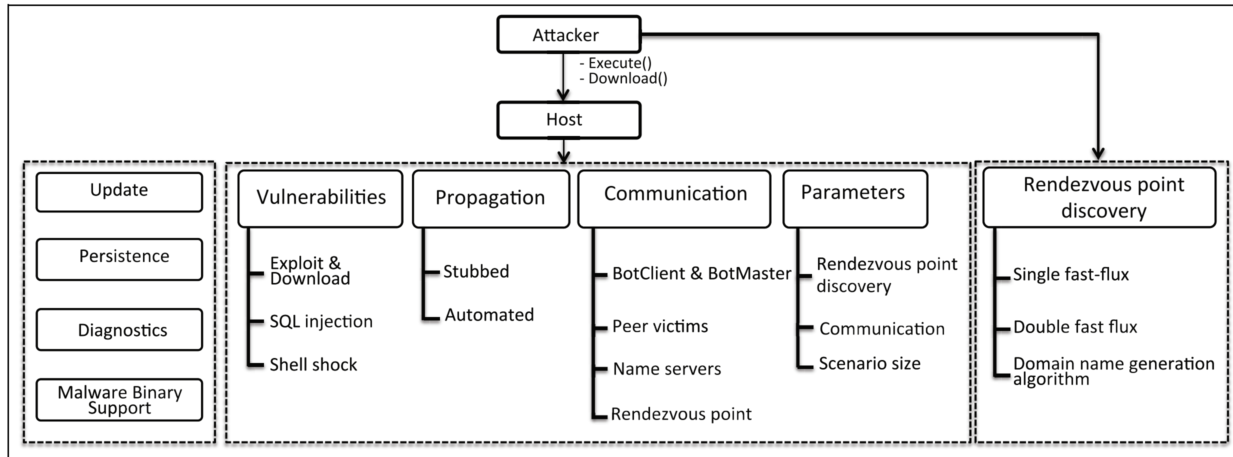

Figure 3. Diagram of the malware design objects.

using NSNAME. If found, it uses the resolved addresses as a private name server to resolve RPNAME and the resolved address as the rendezvous point.

- Domain Name Generation Algorithm (DGA) – BotMaster picks a name for the rendezvous point using an algorithm that deterministically translates the current time (down to hour granularity) into a string. It selects a rendezvous point, starts rendezvous service on that and registers the string with the public name server. In this manner, the string resolves to the chosen rendezvous point. BotClient generates a range of string based on the current time plus and minus an interval and queries the public name server for each string. Note that as if the clocks on the victim and BotMaster are within the range of each other, then the set of strings generated by BotClient contains the string registered by BotMaster. If the public name service resolves any of the strings, BotClient uses the resolved address for that string as the rendezvous point.

We now detail the properties of the rendezvous point that defines the capabilities of the BotMaster.

Rendezvous Point Alteration– BotMaster may periodically stop the current rendezvous service, pick another rendezvous point, start the rendezvous service on that newly chosen rendezvous point and update the public and private name server accordingly. Depending on the time of last rendezvous service migration, the name server update may or may not result in a new string. Stopping the current rendezvous service causes communication failures that prompt BotClients to repeat the rendezvous point discovery steps.

Rendezvous Point Selection– The choice of method to use is controlled by BotMaster. BotClients discover the choice based on the results of public name server queries. If RPNAMEis resolved, then BotClient knows that BotMaster uses single fast flux. If NSNAME is resolved, then BotClient knows that BotMaster is using double fast-flux. If neither of them is resolved, BotClient assumes that BotMaster uses DGA. When BotClient is triggered by communication failures to rediscover the rendezvous point, BotClient forgets the method it had been using. This aims at giving flexibility to BotMaster changing the method dynamically.

Rendezvous Point Failure– If a BotClient does not have access to the public name server infrastructure, none of the methods discover the rendezvous point. Further, intermittent network and software faults may cause either rendezvous point discovery or rendezvous service usage to fail. When BotClient cannot discover a rendezvous service, BotClient resorts to the *parental linage* for communication with BotMaster. When a BotClient runs, it attempts to infect other victims, and the BotClients on these victims are considered the children of infecting BotClient. During the infection step, the identity of the infecting BotClient (the parent) communicates to the new victims (the children).

BotClients can perform *store-and-forward* relay service for their children. They chose to contact BotMaster through their parent's store-and-forward service (if they cannot find an usable rendezvous service as discussed previously). We note that the initial victim has no parent, therefore, if the initial victim cannot use the rendezvous service, the entire botnet will be unable to communicate with BotMaster. If initial victim's access to rendezvous service fails because of an intermittent fault, its store-and-forward capabilities allow subsequent recovery with minimal message lost. However, if the failure is structural (e.g., the initial victim cannot access the required public name server, private name server, and rendezvous point), botnet formation irrecoverably fails.

Rendezvous Point Recovery– In addition to communication failures triggering BotClients to attempt rendezvous service rediscovery, BotClients periodically forget the current mode of BotMaster communications and perform the rendezvous service discovery steps again. Therefore, if a BotClient uses parental lineage communication because of an intermittent fault existing during rendezvous service discovery, BotClient eventually reverts to using rendezvous service after the intermittent fault has cleared.

4.4.2 Propagation. This section presents the steps taken by BotClient to find other victims and propagate to them.

Stubbed Reconnaissance– BotClient consults compiled-in data structures to determine the set of victims and their vulnerabilities. These data structures are prepared ahead of time based on network topology of scenario. We implement a utility program to help this task. The utility program runs from the control computer and determines the set of other computers that can be accessed by each host. It builds a data structure – called the `connectivitySet` – for each scenario using this information. When BotClient is executed, it finds the `connectivitySet` and attacks the computers listed in it. The utility program does not determine which vulnerability to use for each computer; this is specified manually by the creation of a data structure.

Automated Reconnaissance– When using the automated reconnaissance, BotClient uses `nmap` to discover other reachable computers that have the p2paa, SQL database, or open SSH ports. However, the propagation might carry the danger of requiring an extremely long time to find victims. To help ameliorate this risk, at compile time, we use target specification using nmap's host specification format to limit the range of network mapping.

Infection– BotClient attacks computers using one of the three exploits by using the attack method with the given priority order of p2paa, SQL, and shell shock. Infection consists of downloading a copy of BotClient to the victim and executing it. BotClient essentially sends a copy of the executable file to each victim. However, the executable is modified slightly before being sent. The modification includes a particular purpose binary editor built into BotClient, which performs the following steps:

1. Update the parent structure in the executable so that the current computer is listed as a parent. This enables infected computers to know who their parent is in the case of parental lineage communication is used.
2. Add all of the computers that the infecting BotClient has discovered as new victims to the victim's list. This optimization aims at preventing attempts to infect already infected computers. BotClients do not attempt to infect computer listed in the known victim's list. This is enabled by a mutex to prevent multiple instantiations.

Architecture Dependency– Experiment setting needs to support the platforms that a victim uses. That is, if BotClient executables are compiled for a particular architecture (i.e., different operating systems), it cannot be executed on the other architectures. Therefore, BotClient attempts to download each version to each victim and tries to run both executables, knowing that only one succeeds. An approach in which the infected host determines the platform of the victim and downloads only the appropriate version may appear more optimal, however, since the victim's reconnaissance may discover new victims of any platform, it is always necessary to download both versions.

4.4.3 Communication. The BotMaster use various communication channels to make the botnets resilient to blocking or shutdown. Therefore, it is vital to consider the range of C2 topologies and communication protocols that attackers typically use. This section discusses communication channels required for the botnet lifecycle as presented in Table 1.

All communications between BotClients, rendezvous services, and BotMaster may prefer TCP, UDP or both. For instance, BotClients and rendezvous service interface may use port 80 to hide in HTTP traffic. The rendezvous point knows the address of the BotMaster through a

Table 1. Malware communication channels.

Communication
Status messages from BotClients to BotMaster
Execute command messages from BotMaster to BotClients
Download command messages from BotMaster to BotClients
Inject attack messages from BotClients to p2paa on peer victims
BotMaster and BotClient interaction with public and private name servers
BotMaster and BotClient interaction with public and private name servers
BotMaster starting and stopping rendezvous service on chosen rendezvous point and private name server

command line argument. Parental lineage communication occurs over a specific port number in the upward direction (child to parent), and another port number is assigned in the downward direction (parent to child). Inject attack messages occur on the p2paa port. The port numbers are configurable, as dynamic port numbers can be assigned to subsequent versions.

BotClients and BotMaster Communication – Communication between BotMaster and BotClient is BotClient driven. BotClients periodically send a status message to BotMaster. Upon receipt of the status message BotMaster responds with an acknowledgment message that may contain a BotMaster command `execute()` or `download()`. The acknowledgment contains the BotMaster command if the attacker has entered a command that has not yet been sent to the specific BotClient. This is indicated by a command serial number included in the status message.

Results of `execute()` and `download()` commands are conveyed from BotClient to BotMaster in the status messages. It parses these fields out of the status message and displays them on the BotClients Xterminal. An one-time result is supplied for download commands, which indicates that the download has completed. For `execute()` command multiple results occurring over a considerable time span can be reported through status messages depending on the executed command type. For instance, `free -s 1` command produce a status message every second, whereas `free` command with no arguments provides a single status message. When the commands are used to convey download or execute results, BotClients generate status messages on-demand, not strictly periodically.

BotMaster and Name Server Communication – BotMaster interaction with public name server is through the Linux utility `nsupdate`, which is used to add and remove entries from the public name server. It defines a temporary file corresponding to `nsupdate`'s specification and executes nsupdate command with that file as an argument.

BotMaster interaction with the private name server is through the `scp` (Secure Copy), `ssh` (Secure SHell) and `Dnsmasq` utilities. `Dnsmasq` (DNS forwarder and Dynamic Host Configuration Protocol (DHCP) server) is a lightweight private name server used by BotMaster. It is configured to resolve a single name through a configuration file (which is identical in format to the standard `/etc/hosts` file). BotMaster constructs the necessary file, uses `scp` command to connect to the private name server, and uses `ssh` command to that computer to start `Dnsmasq` by specifying the file(s) as an argument. When migrating to a different name server, it connects to the current name server and uses `pkill()` to stop the running `Dnsmasq`.

BotClient interaction with the public and private name server is implemented through the `getaddrinfo()` library call and Linux utility `host`, respectively. When attempting to resolve a set of DGA names, BotClient uses multiple threads to perform queries in parallel, since queries are mostly I/O bound. Depending on the communication characteristics, the number of threads is adjusted to achieve reasonable network and name server utilization (controlled by a manifest constant in BotClient). For instance, too many threads can congest slow networks or low powered name servers, and too few threads can result in excessive DGA rendezvous point discoveries.

4.4.4 Parameters. While our implementation covers the design of the real-world malware, it is necessary to have a parameter space that governs its behavior and activities. We now elaborate these parameters. First, malware operation requires internal manifest constants that are set to reasonable values for the experimentation. However, the malware behavior can be changed dynamically by customizing the parameters in the manifest file. We categorize these parameters as (1) rendezvous point discovery, (2) communication, and (3) scenario size. Table 2 presents the list of the parameters. The botnet executables need to be re-built for parameter changes to take effect. As a convenience, a few of the parameters that the operator may be more likely to change can be dynamically configured through the use of configuration file at start up.

We remark that much real-world malware shares the similar parameters. These parameters provide a valuable perspective to their analysis. For instance, Zeus crimeware toolkit employs both static and dynamic configuration files. These files instruct an infected host to change its behavior for becoming stealthy and efficiently facilitate its criminal activities. We observe that Zeus includes similar parameters as we have identified in our implementation. For instance, it uses `timer_logs` parameter to specify the time interval to upload the stolen data on infected hosts. More advanced parameters such as `file_webinjects` is also used. It includes a list of HTML code with domain names that is used for stealing credentials.[31]

4.5 Persistence

Persistence makes the BotClient start automatically after a previously infected node is rebooted. While most malware attempts to achieve persistence, persistence can often be a problem in an experimentation environment unless carefully monitored, and this may result in inadvertent malware execution. Persistence is by default dynamically configurable. If persistence is enabled through a dynamic configuration option after startup, BotClient attempts to edit the configuration file on the infected node to add itself as a program

Table 2. Parameters used to govern the botnet behavior.

Rendezvous Point Discovery Parameters	
DOMAINLEN	Length of the generated name for DGA-based botnets
TLD	Top-level domain for all names queried or registered with the name server
RANGE	Number of years/months/days/hours/minutes +/- the current year for generating the set of DGA names for DGA BotClient
NSNAME	For double fast flux specifies the name of the private name server registered with the public name server
RPNAME	For single and double fast flux specifies the name of the rendezvous point registered with the public (single fast flux) or private (double fast flux) name server.
MAXT	Number of parallel threads used for name server queries for DGA botnets (High values may tend to flood network and name server, low values increase the rendezvous point discovery time)
RPTTL	Time-to-live in seconds for records added to the public and private name server, and the period at which rendezvous service is migrated. (For double fast flux, the private name server is migrated every cycle through the rendezvous points)
MAXRPS	Maximum number of rendezvous points the BotMaster migrates the rendezvous service among
MAXNSS	Maximum number of private name server computers the BotMaster migrates the private name server among
Communication Parameters	
MIN_MAX_STATUS	Time period of an infected host that attempts to send a status message to the rendezvous point
MAX_BOTNET_ROUTES	Maximum number of parents between BotMaster and BotClient for parental lineage communication
CANDCUPPORT	Port number used for child to parent parental lineage communications
CANDCDOWNPORT	Port number used for parent to child parental lineage communications
MAX_COMMANDS	Maximum number of commands from BotMaster to BotClients that may be stored awaiting delivery through parental lineage communications
MAX_COMMAND_FDS	The largest file descriptor that is used for the pipe in executing BotMaster issued commands
MAX_REPLIES	Maximum number of command replies that may be buffered awaiting delivery from BotClient to BotMaster
Scenario Size Parameters	
MAX_PARENTS	Maximum number of parents any BotClient binary edits into the BotClient executable during propagation
MAX_KNOWN_VICTIMS	Maximum number of victims the BotClient binary edits into the BotClient executable during propagation
MAX_VICTIMS	Maximum number of victims BotClient attacks
MAX_T	Maximum number of concurrent threads that the rendezvous service may use for BotClient to BotMaster communications
MAX_V	Maximum number of BotClients the BotMaster can support
MAX_COMMANDS	Maximum number of BotMaster commands the BotMaster stores that is pending delivery to BotClients

that automatically starts after reboot. Persistence may not always be successful since some infection vectors do not obtain root privilege on the infected node. This is due to the reason that root privilege is required to edit the configuration file. We create a script allowing to run a BotClient code after an infected host reboots.

4.6 Diagnostics

Testbed problems, botnet configurations, and operator errors may result in execution failure of a BotClient. Therefore, it is necessary to specify a correct diagnostics to resolve such problems. We write logs to a file for all components upon encountering an unexpected situation. Each component writes a message to `stderr`, which includes the C function name and line number, a human readable description of the situation, and a human readable rendering of any system error codes. Further, we write the description of other significant non-error events (e.g., rendezvous service migration) to `stderr`. When botnet code starts the botnet component, the stderr stream of each component is redirected into a file specific to that component of P2paa, BotClient, BotMaster and rendezvous point for quickly tracing back to the errors.

4.7 Other design issues

Extensibility Support– A typical use of a botnet is to download and execute additional malicious code on the

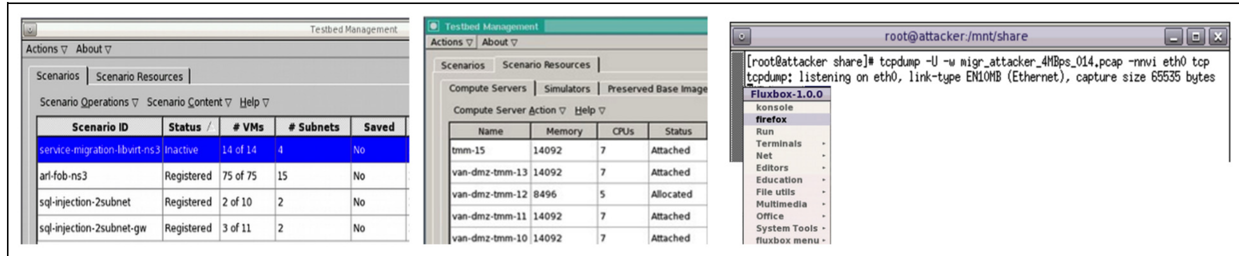

Figure 4. User interface showing the scenario, scenario resources, a connection to a host for data collection and a command window enabling various operations on the host machine.

botnet victims. Any number of code can be saved under a sub-directory of a BotCliet and the code is then can be executed. We detail exfilClient malware that we integrate to the BotClients. ExfilClient is a C program compiled into a binary executable that performs exfiltration by searching files on the victim. It looks for text files containing the sensitive words such as ''secret'' and upon finding such a file, outputs the entire file to stdout.

ExfilClient is executed through the botnet by first downloading (using the `download() command`) it from the attacker computer to the botnet victims. Then `execute()` command is used to run it on the botnet victims. This results exfiltrated files being displayed in real time on the BotClient xterms. It is also recorded in the individual log files on the attacker's computer to extract the exfiltrated files contents at the attackers' convenience. ExfilClient is built by the make, and the resultant executable exfilClient is distributed to `/tmp` on the attacker's computer. BotMaster's download command can be used to distribute it to an arbitrary location on the botnet victim computers (e.g., download `/tmp/exfilClient`, `/tmp/ex`). Then, it can be executed using execute command `/tmp/ex`.

User Interface and Usability– To hide the complexity of using the capabilities described previously, there is a need for an automated toolkit to reduce the burden on the experimenter. Therefore, we developed graphical user interfaces (GUI), which is a pre-packaged experimental environment that let a user load and modify the malware binaries and monitor the errors. Figure 4 presents examples of interfaces allowing users to run malware, collect data and enable its parameter tuning. The interface also allows configuring the each experiment. An experimented is able to define malware parameters and employ previously discussed parameters in an automated way.

5 Validation

In this section, we present a validation procedure to analyze the parameterized malware behavior. Dealing with the parameters is a challenging problem, but ignoring them may lead evaluations to provide misleading evidence. Therefore, a complete, error-free experimentation for this problem requires additional analysis. The procedure aims at fostering the evaluation of detection techniques and preventing erroneous experimentation. Additionally, we present generating legitimate traffic from synthetic tools, and publicly available traces to blend in malware traces for analyzing their traffic characteristics.

5.1 Validation procedure

We introduce a validation procedure to overcome experimentation and parameter variation fallacies while operating the malware. We note that even though the malware is programmed correctly and generating no error logs, a parameter value variation or erroneous experimentation may have an impact on evaluation results. To overcome these issues, we present a sensitivity analysis of (1) raw files, and (2) fine-grained behaviors. Figure 5 illustrates the seven steps of the validation procedure. We next describe these steps.

1. **Experimental design:** We run malware on a CyberVAN testbed with a various number of interconnected hosts and network topology. We terminate experiments after at least one hour of data collection. All reported measurements are obtained from infected and benign hosts, as this is the portion that our framework attempts to validate.
2. **System and network level data collection**: We collect system and network level raw files from a subset of hosts. Our goal is having a nearly comprehensive snapshot of the host. Using a subset of the hosts may affect some of our assessments. For instance, we may miss a portion of the network trace that includes an abnormal behavior, and this fraction of traffic may change the resulting evaluation. We address this problem in step 4 below.
3. **Error logging**: Error logs contain the critical error records obtained from design components while

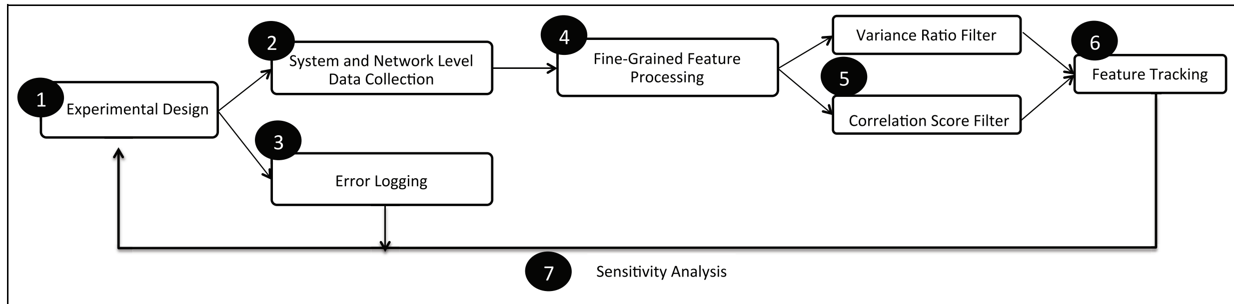

Figure 5. Validation procedure: The steps are used to point and analyze the parameterized behavior of the malware.

running the malware. We write the description of other significant non-error events to track the malware behavior. These errors help us to map a record and the component that causes the error.

4. **Fine-grained behavior processing**: To have a comprehensive view of the malware behavior, we collect fine-grained behaviors both from network-level and system-level activity. Network level exploration includes inter and intra-flow based attributes.[32–35] These attributes are selected based on the understanding of packet size distribution, timing based statistics (e.g., packet inter-arrivals), and packet/byte transfer rates. We also use DNS-based features to reveal the DNS request behavior, e.g., number of IP addresses, number of distinct countries, and TTL value statistics. The system-level properties – system state and running processes activity of file system, memory, network, and registry – are recorded with the sysdig chisels that unifies the results of system tools such as strace, tcpdump, htop, iftop, and lsof.[36] A reader can refer to complete list of behaviors: argus flow analyzer for network level,[1] sysdig filters for host-level,[2] and tshark for DNS filters.[3]
5. **Filter**: The malware behavior is compared with the benign behavior based on filtering the characteristics of malware and benign ground-truths. We use correlation score filtering algorithm[37] and variance ratio.[38] The aggregated score of these algorithms gives the relevance of each feature as having a minimal correlation between behaviors but being highly correlated to a specific behavior. Specifically, this helps to find a subset of attributes that are identified as the most statistically different ones between the malware and benign sample behaviors.
6. **Feature tracking**: This component is used to aggregate the results of correlation scores. The results are associated with parameters that are leading inter-class differences. We use this information to validate the parameter space and implementation errors caused by the attribute variability between malware and benign samples.
7. **Sensitivity analysis**: The scores given by feature tracking component is used to repeat the experiments. We change the parameters and start a new independent experiment from step 1. We then explore the parameter space to observe the effect on the subsequent running outputs with the new parameters. This process pinpoints the source of significant differences between benign and malware behaviors and allows for analysis of inconsistent or unclear behaviors or depict (unrealistically) optimistic detection results.

We build a Random Forest classifier (RF) using the attributes extracted at filtering step (step 5) for evaluating the malware detection performance. We note that our goal is neither optimizing feature selection nor detection technique. We would like to observe the classifier's response to the fine-grained feature space of the benign and malware ground-truths with various parameter settings. We note that the outputs from feature tracking component enable to repeat a broad range of scenarios. This allows us to understand the impact of parameters on the malware behavior and its detection results. In addition to the synthetic traffic, we use the real-world traces and follow same validation procedure. This uncovers the behavioral differences of the malware observed through benign real-world traces performed at filtering step.

5.2 Legitimate traffic generation

It is a common practice to mix benign and malware traffic to separate malicious and legitimate activities. We study two common methodologies to generate legitimate traffic

[1]http://nsmwiki.org/index.php?title=Argus
[2]http://www.sysdig.org/wiki/sysdig-userguide/#filtering
[3]https://www.wireshark.org/docs/dfref/d/dns.html

to compare their network characteristics with the malware traffic: (1) traffic generation through synthetic modeling of users based on their behaviors, (2) using real-world traffic. We use these traffics to observe and find solutions for the artifacts after blending with malware traces in the following section.

5.2.1 Modeling users. Our first approach is based on controlled traffic generation tool called Markov Brain. Markov Brain generates synthetic network traffic that is captured from a real computing environment. It supports hundreds or thousands of unique user profiles. We use Markov Transition Matrix (MTM) for the set of commands for a given user, where entries in the matrix encode probabilities of switching from one user activity to another. The major strength of the approach is generating MTMs for different user behaviors. Existing systems attempt to fill out the transition probabilities randomly. Instead, we maintain a correspondence between the entries and transition probabilities estimated from actual users. The major shortcoming of this approach is the difficulties in modeling the user behavior convergence and user differentiation.

To address these problems, we start with the user command frequency distribution π_0. π_0 is the principal eigenvector of the MTM and is thereby an important characteristic since the eigenvector points to the long-term behavior of the user. By controlling the distance between π_0 for different users, we have a reliable way of ensuring that this process prevents coalescing into the same user over time.

The next problem is choosing π_0 often for hundreds of users. The correct values of π_0 depends on the accurate extraction of system process usage characteristics. With this information, we define a closed-loop process that produces a simulated user who, in the long term, is similar to the real user. The process includes three steps: (i) we extract process usage statistics from real user activities; (ii) we adjust π_0 values for simulated user, and (iii) we configure the MTM based on the new π_0 values. This approach allows a simulated pool of users that follows a group of real users' behavior so that the application usage mimics the real user activity.

5.2.2 Background traffic. We discussed that Markov Brain is able to generate traffic from the actions of the modeled users. However, there is a need for using realistic background traffic that represents the traffic volumes seen by network elements such as servers or firewalls. The generation of the background traffic may affect the real network packet distributions, and packet inter-arrival times. To solve this problem, we use the recently released DETER's Lego-TG (Lego Traffic Generator) toolset.[39] Lego-TG enables observation and modeling of real-world traffic for future replay. It distinguishes itself from the prior art such as TCPreplay, swing, bit-twist, and harpoon, by decomposing traffic generation into multiple Lego-style blocks. Depending on the features of the traffic to be generated, the blocks are used to aggregate traffic that matches the desired background traffic.

5.2.3 Real-world traffic. One of the most important aspects of the network-level malware detection is blending the malware traffic with the real-world traffic. This process needs caution, as the packet traces may lack some application types. For instance, malware mimicking the FTP traffic that is not found in legitimate traffic may yield unrealistic results. To avoid such artifacts, we use a small scale organization network recorded at the University of Twente with around 35 employees and over 100 students.[40] Further, we use Lawrence Berkeley National Laboratory (LBNL) traces that aim at capturing the characteristics of the traffic patterns and applications recorded in an enterprise network.[41]

6 Results

This section presents our findings while operating the malware in the testbed. Figure 6 shows a snapshot of analysis results with a specific parameter configuration of the malware. The x-axis presents the 13 fine-grained behaviors, and y-axis presents the frequency of that behavior ranked as the most discriminating between malware and benign ground-truths collected from multiple hosts (recall the validation steps in the previous section). Table 4 describes the 13 behaviors. We repeat the experiments by changing the parameters. As shown in the right histogram plot, the scores given by feature tracking is different for some attributes. The resulting list of features is one of the many examples used to pinpoint the fine-grained behaviors that may cause this difference based on the parameter variations, and consequently are the reasons for high detection rates.

In this way, we find the attributes yield a discriminative behavior through parameters and allow systems to detect the malware easily. These signals may help detect the malware, yet they might not be the generalized malware behavior under various parameter configurations. Therefore, an analysis of complete parameter space of malware is required to observe the complete behavior of the malware through different parameter settings. Our experiments intended to outline the challenging nature of this problem. We next discuss some of our results that we observe during the parameterized malware operation in the testbed.

Wrong port configuration easily reveals the infected hosts: We begin our evaluation with a simpler case in

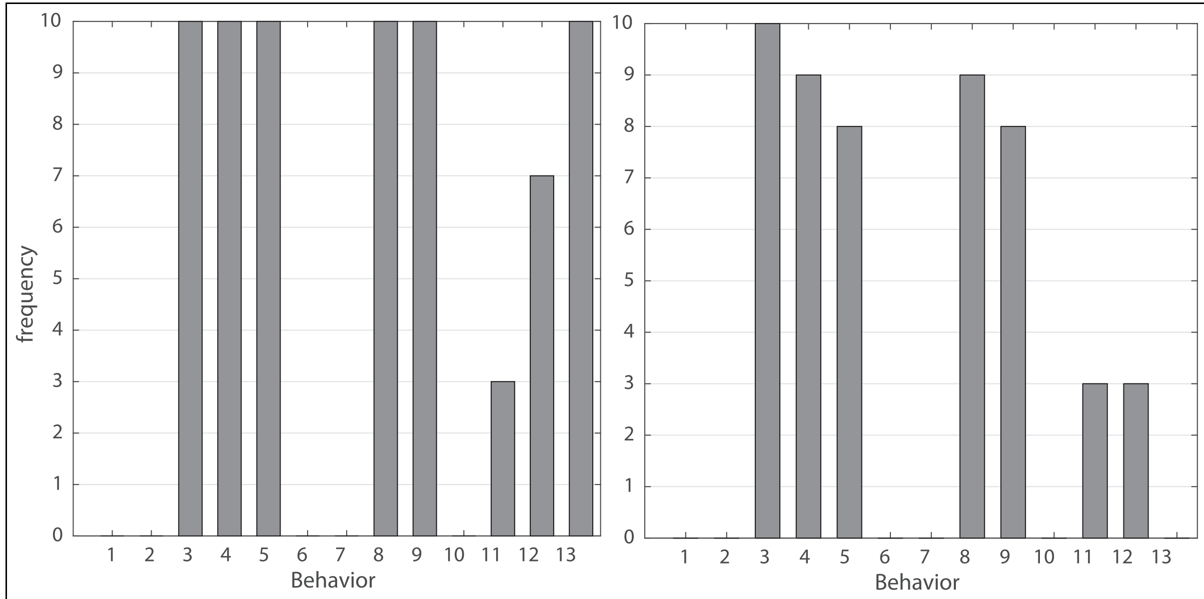

Figure 6. Sensitivity analysis of parameters: (Right) Two figure shows the feature variations between legitimate and malware ground-truth through parameter configurations. The 13 behaviors used for this analysis are listed in Table 4.

which we found that the detection mechanism may always produce the perfect separation between the benign and malware samples. In this set of evaluations, we salted the benign traffic with the malware traffic and attempted to detect the malware. We found that the main reason for the perfect results is the port numbers used in the feature space. This leads us to analyze the raw packet traces for the application categories and their protocols. We found that benign traffic does not include any port number associated with any of the port numbers used by the malware (see port number parameters of CANDCUPPORT and CANDCDOWNPORT configurations). Therefore, malware samples are perfectly detected. However, the use of port numbers as features might cause serious problems in real systems, as malware may use the port numbers of legitimate applications to obfuscate their activity and pass through firewalls that block certain port number ranges.[42]

Increasing the application diversity camouflages malware activity: In our second set of experiments, we found that elimination of port numbers from the feature space increases the detection error from 0% to 3.2% which is high for a basic detection system. We rerun the experiments by increasing the application diversity using the LBNL enterprise network traffic. Table 3 presents the protocol breakdown and corresponding applications of the benign traffic composition. The detection error increases to 12.4%. We found that malware packets are mostly classified as benign HTTP(S) traffic and vice versa. To better understand the types of applications that might camouflage malware, we run the experiments after blending with the simpleWeb traffic. The detection error yields 9.2%. We observed that simpleWeb traces include a vast variety of chat services and less HTTP traffic that is different than LBNL traces. Therefore, it is necessary to evaluate malware behavior by considering the various application types.

Table 3. Legitimate traffic applications and corresponding protocol diversity in LBNL enterprise network traffic.[41] Without such variety, the malware traffic can be easily detected.

App.	Protocols
Backup	dantz, veritas, connected-backup
Bulk	FTP, HPSS
Email	SMTP, IMAP4, IMAP/S,POP3, POP/S, LDAP
Interactive	SSH, telnet, rlogin, X11
name-srv	DNS, netbios-NS, SrvLoc
net-file	NFS, NCP
net-mgnt	DHCP, ident, NTP, SNMP, NAV-ping, SAP, NetInfo-local
Web	HTTP, HTTPS
Windows	CIFS/SMB,DCE/RPC, Netbios-SSN, Netbios-DGM
Misc	Steltor, MetaSys, LPD, IPP, ORACLE-SQL, MS-SQL

Unnatural timing heterogeneity easily reveals the malware: In this set of experiments; we mix the packet traces taken from malware execution and the real-world network traffic. The feature tracking process reports the timing based behaviors as the most discriminating ones after

Table 4. An example of behaviors used for parameter analysis.

#	Description
1	Average number of packets client to server
2	Average number of packets server to client
3	Average number of unique destination ports
4	Time to live (TTL) of DNS packets
5	Round Trip Times (RTT)
6	Total number of bytes
7	Average jitter client to server
8	Average jitter server to client
9	Flow duration
10	Events for a specific process
11	Average number of distinct IPs in DNS packet
12	Protocol
13	Average number of NXDOMAIN responses

elimination of port numbers. The timing-based features are calculated based on the flow duration and goodput. One might wonder at this point whether we restrict ourselves to a set of representations that are too specific to be useful in a malware detector. However, these features explicitly include the inter-arrival time and Round Trip Times (RTT) of the packets and show a positive impact on any detection system.[16] Figure 7 presents the RTT samples of an example malware traffic and real-world traces. We found that the malware timing-based features are significantly shorter than those of benign traffic. The long delay of benign communication could be due to several factors. For instance, the stepping-stones of real-world links exhibit high sensitivity to the bandwidth, connectivity, or processing latency. To solve this problem, the unnatural heterogeneity between timing characteristics of malware and benign traffic needs to be calibrated using the samples in real-world traffic to prevent inadvertent impacts on the detection results.[16,43]

Parameter tuning decreases the malware signal: In these set of experiments; we show how misconfiguration of Non-existent Internet Domain Names (NXDOMAIN responses in DNS packets) could easily reveal the infected hosts. We first predict the number of NXDOMAINs of hosts to observe the domain name generation (DGA) algorithm used by the malware. This allows infected hosts to evade the blacklists and signatures. Our goal is to find the differences between legitimate DNS queries generated by the legitimate users via application typos or errors and those caused by the malware. To identify an appropriate threshold for the number of NXDOMAINs, we examine hosts over a time interval and count the number of such domains per minute. Our intuition is to have the benign hosts to be on the threshold of the NXDOMAIN responses recorded at training time. We observe that infected hosts tend to query large sets of domain names, yet few hosts are able to resolve to the IP addresses of the C2 servers

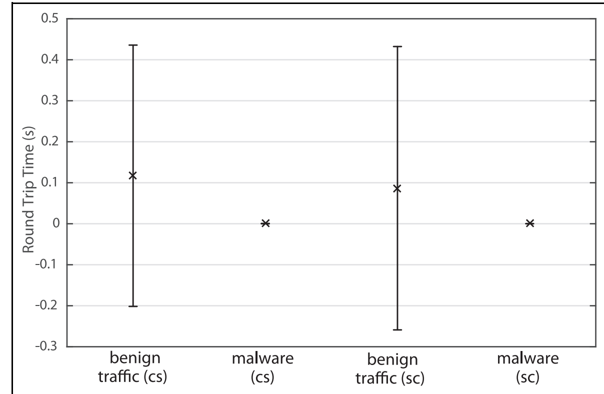

Figure 7. Comparison of the unnatural packet timing heterogeneity between malware and legitimate traffic: Client to server and server to client RTTs inadvertently signals malware.

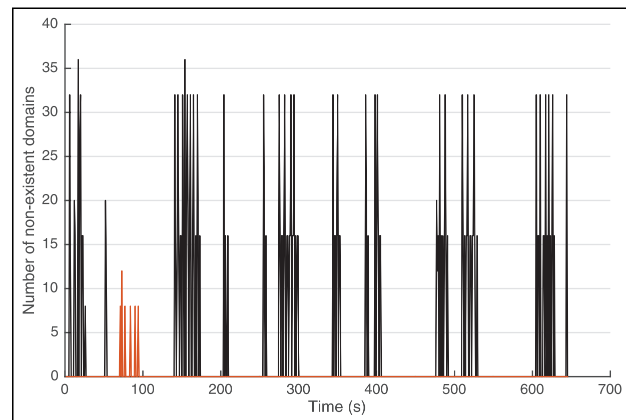

Figure 8. Lack of parameter tuning in rendezvous point discovery reveals the malware behavior: Infected machines generate similar patterns over time to connect a rendezvous point while attacker fails to register the domain (black). The similar behavior for the benign traffic is illustrated between 60 and 90 seconds.

successfully. After an investigation, we found that the parameter `MAXRPS` is configured incorrectly in the malware code and it causes an outlier of non-existing domain names. Figure 8 shows the parameter tuning impact on the malicious and benign hosts. We find that few queries of infected hosts to the rendezvous point are successful. However, frequency-based statistical detection might be ineffective to observe this behavior at a low number of NXDOMAIN responses due to the low frequency of NXDOMAIN responses generated by the infected hosts.

Lack of user interaction in malware infection: It is essential to include user interaction mechanisms in malware execution traces based on the malware type. In these

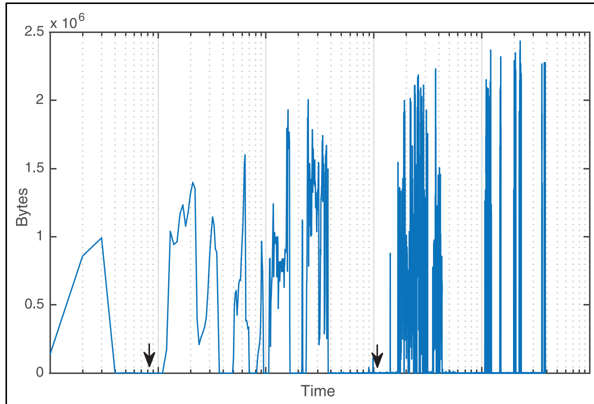

Figure 9. Initial botnet infection involves no end-user interaction: The arrows show the infection times of a host. Well-designed experiments needs to consider user interaction depending on the malware type.

experiments, we focus on how infected hosts download and install the malware source code. We found that the infected hosts lack user interaction at the infection phase. More specifically, the initial infection occurs directly after running the malware code at hosts, i.e., an attacker does not trick users into downloading and executing the BotClient software. This process may not be valid in real-world captured malware traces. For instance, attackers may send email messages that make users install the attached malware programs or trick users into visiting a malicious URL. Figure 9 shows the malware infection times with arrows and the closest observation time of bytes that shows the transmitted data in a log scale time interval. Analyzing log records, we find that the packets are generated as expected after infection, yet the infection occurs without any packet transmissions, and infected hosts directly start the malicious activity (i.e., exfiltrating sensitive data). This limits the correct observation of system and network behaviors.

7 Recommendations

Throughout this work, we showed a malware design and parameters governing its behavior. We outlined a validation methodology and pointed out the fallacies made possible by the abuse of parameter configuration. Space limits do not allow us to present our measurements exhaustively. Instead, in this section, we present recommendations, both nontechnical and technical, on malware modeling and experimentation. Our recommendations outline the operational and experimental suggestions emerged from this work with the hope that they will be helpful to the researchers.

1. Malware modeling and experimentation is crucial for evaluation of detection systems. In order to have realistic results, its operation should include an understanding of implementation details which provides ground for its reproducibility and validation. For instance, before presenting the evaluation results, parameters of its execution and capabilities of the attacker should be undoubtedly presented.
2. Malware detection and remediation techniques are effective when they present the assumptions of the malware operation. In many cases, it may not be possible to create a system to detect the complete behavior of the malware. However, experimentation must discuss whether simplifying assumptions changes the detection results in a meaningful way or not.
3. The value of the experimentation results depends on the exploration of malware behavior through its execution with various parameters. The impact of these parameters should not be evaluated with constant parameters. Observation of malicious behaviors through various parameter configurations is essential for understanding its complete behavior over time.
4. The use of honeypot traces or executing malware binaries that are obtained from malware repositories may limit the scope of malware execution to particular software and hardware architectures. However, the execution of malware must be evaluated against reproducibility of its execution in different software and hardware. This is often possible with its implementation and execution in the testbed facilities.
5. Another problem of using ready-to-use malware binaries is the lack of characterizing the emerging or new variants of the malware. Malware implementation in a controlled environment may mitigate this issue and help for proactive experimentation of uncovering new attack vectors in subsequent malware variants.
6. While evaluating malware traffic detection, synthetic malware traces and real-world traffic traces should be blended properly. That is, it should ensure that statistical consistency between traces is identified and calibrated. For instance, lab experimentations miss the processing overhead and latency of stepping-stones found in the real network traffic.
7. Another important point with mixing the legitimate traces with malware traffic traces is that real world traces must include various applications obtained from both academic and enterprise networks. This is due to the reason that application

types may impact the detection of malware traffic from legitimate applications.
8. Finally, malware design and experimentation require substantial human resources and time. Researchers should release pre-packaged implementations and datasets publicly so that others may benefit from them.

8 Related work

Emulab,[8] PlanetLab,[9] and DETER[10] are the testbed experimentation facilities enabling malware implementation. Our design process and validation procedures can be easily integrated into these platforms.

Our work is related to a number of works in malware modeling and experimentation. Barford et al. presented initial steps toward building a flexible and scalable laboratory testbed for botnets.[12] Calvet et al. presented at-scale botnet emulation in laboratory conditions and specifically analyzed Waledac botnet.[44] Allodi et al. presented MalwareLab which is an isolated environment for testing the exploit kits leaked from the markets.[11] John et al. created Botlab to monitor the behavior of spam-oriented bots.[45] Alwabel et al. provided a discussion of safe and automated malware experimentation.[46] Our goal is different from these studies. We seek to build a flexible malware similar to in-the-wild malware based on reverse engineering of its parameterized behavior. Other previous works have attempted to emulate the malware. For instance, Lee et al. and Jackson et al. focused on emulation of the botnet protocols.[47,48] These works are limited to communication patterns. Therefore, they do not capture the complete malware behavior. Instead, we analyze the sensitivity of malware operation using various parameters and analyze the parameters that may cause erroneous experimentation.

9 Conclusion

We have presented a complete implementation of a botnet command and control network using the techniques that are in use by current botnets. The implementation supports general capabilities that allow attackers to execute arbitrary commands on the infected botnet victims. The implementation is flexible; it is written to allow easy integration of additional exploits and vulnerabilities. Operating the synthetic botnet in a testbed, we quantify and characterize the malware operation by application of sensitivity analysis based validation. Such analysis aims at fostering the critical evaluation of proposed detection techniques and stymie unintentionally erroneous experimentation. We found that both design pitfalls and parameter configurations are an indicator of the inadvertent mistakes in experiments. These findings demonstrate that if parameter variations are not explored, these variations may lead to overly optimistic conclusions and detection systems that are ineffective in practice. Finally, to help the research community with their malware experiments, we have devised operational and experimental recommendations. In the future, we will explore a wide range of malware models and evaluate their parameterized behavior to generalize our approach beyond the botnet case study.

Declaration of conflicting interests

The authors declared the following potential conflicts of interest with respect to the research, authorship, and/or publication of this article: Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

1. Livadas C, Walsh R, Lapsley D, et al. Using machine learning techniques to identify botnet traffic. In: *Local computer networks*. Piscataway, NJ: IEEE, 2006, pp.967–974.
2. Strayer WT, Lapsely D, Walsh R, et al. Botnet detection based on network behavior. In: *Botnet detection*. Berlin: Springer, 2008, p.1–24.
3. Gu G, Perdisci R, Zhang J, et al. BotMiner: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In: *USENIX Security Symposium*, 2008, pp.139–154. Berkeley, CA: Usenix.
4. Garćıa S, Grill M, Stiborek J, et al. An empirical comparison of botnet detection methods. *Comput Secur* 2014; 45: 100–123.
5. The Honeynet Project & Research Alliance, Know your Enemy: Tracking Botnets http://www.honeynet.org/ (2015, accessed 10 April 2016).
6. ANUBIS. http://anubis.iseclab.org (2015, accessed 10 April 2016).
7. Wurzinger P, Bilge L, Holz T, et al. Automatically generating models for botnet detection. In: *Computer Security – ESORICS 2009*. Berlin: Springer, 2009.

8. White B, Lepreau J, Stoller L, et al. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 2002.
9. Chun B, Culler D, Roscoe T, et al. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review* 2003, p.3–12.
10. Benzel T, Braden R, Kim D, et al. Experience with deter: a testbed for security research. In: *Testbeds and research infrastructures for the development of networks and communities*. Piscataway, NJ: IEEE, 2006.
11. Allodi L, Kotov V and Massacci F. MalwareLab: Experimentation with cybercrime attack tools. In: *CSET' 13, 6th Workshop on Cyber Security Experimentation and Test*, 2013. Berkeley, CA: Usenix.
12. Barford P and Blodgett M. Toward Botnet Mesocosms. Usenix workshop on HotBots. 2007, p.6–6.
13. Binsalleeh H, Ormerod T, Boukhtouta A, et al. On the analysis of the zeus botnet crimeware toolkit. In: *Privacy security and trust*. Piscataway, NJ: IEEE, 2010.
14. Wyke J. Duping the machine – malware strategies, post sandbox detection, https://www.virusbulletin.com/ (2016, accessed 20 April 2015).
15. Garćıa S, Zunino A and Campo M. Survey on network-based botnet detection methods. *Secur Commun Networks* 2014.
16. Celik ZB, Raghuram J, Kesidis G, et al. Salting public traces with attack traffic to test flow classifiers. In: *Cyber security experimentation and test (CSET)*. Usenix; 2011.
17. Celik ZB, McDaniel P, Walls R, et al. Malware traffic detection using tamper resistant features. In: *Military communications conference (MILCOM)*. Piscataway, NJ: IEEE, 2015.
18. Celik ZB and Oktug S. Detection of fast-flux networks using various dns feature sets. In: *Computers and communications (ISCC)*. Piscataway, NJ: IEEE, 2013, p.000868–000873.
19. Celik ZB, McDaniel P, Izmailov R, et al. Extending detection with forensic information. arXiv preprint arXiv: 160309638. 2016, https://arxiv.org/pdf/1603.09638v3.pdf.
20. Celik ZB, McDaniel P and Izmailov R. Feature cultivation in privileged information-augmented detection. In: *Proceedings of international workshop on security and privacy analytics* (CODASPY IWSPA) (Invited Paper), 2017.
21. Andriesse D, Rossow C, Stone-Gross B, et al. Highly resilient peer-to-peer botnets are here: An analysis of Gameover Zeus. In: *Malicious and unwanted software*. Piscataway, NJ: IEEE, 2013, pp.116–123.
22. Stone-Gross B, Cova M, Cavallaro L, et al. Your botnet is my botnet: analysis of a botnet takeover. In: *Computer and communications security*. New York: ACM, 2009, pp. 635–647.
23. Bayer U, Habibi I, Balzarotti D, et al. A view on current malware behaviors. In: *LEET*, 2009.
24. Kotz D, Newport C, Gray RS, et al. Experimental evaluation of wireless simulation assumptions. In: *International symposium on modeling, analysis and simulation of wireless and mobile systems*. New York: ACM, 2004, pp.78–82.
25. Botta A, Dainotti A and Pescapé A. Do you trust your software-based traffic generator? *Communications Magazine* 2010.
26. Aviv AJ and Haeberlen A. Challenges in experimenting with Botnet detection systems. In: *Proceedings CSET'11 proceedings of the 4th conference on Cyber security experimentation and test*, San Francisco, CA, 2011. New York: ACM.
27. Serban C, Poylisher A and Chiang J. Virtual ad hoc network testbeds for network-aware applications. In: *Network operations and management symposium (NOMS)*, 19–23 April 2010, pp.432–439. Piscataway, NJ: IEEE.
28. Berman M, Chase JS, Landweber L, et al. GENI: A federated testbed for innovative network experiments. *Comput Networks* 2014.
29. Ferguson B, Tall A and Olsen D. National cyber range overview. In: *Military communications conference (MILCOM), 2014 IEEE*. Piscataway, NJ: IEEE, 2014, pp.123–128.
30. Chadha R, Bowen T, Chiang CYJ, et al. CyberVAN: A Cyber security virtual assured network testbed. In: *IEEE Military Communications Conference*, Piscataway: IEEE, 2016.
31. Falliere N and Chien E. Zeus: King of the bots. Mountain View, CA: Symantec Security Response, 2009.
32. Bilge L, Balzarotti D, Robertson W, et al. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In: *Computer security applications conference*. New York: ACM, 2012, pp.129–138.
33. Bilge L, Kirda E, Kruegel C, et al. EXPOSURE: Finding malicious domains using passive DNS Analysis. In: *NDSS 2011, 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, 6–9 February 2011. Reston, VA: Internet Society.
34. Li W, Canini M, Moore AW, et al. Efficient application identification and the temporal and spatial stability of classification schema. *Comput Networks* 2009, 790–809.
35. Moore A, Zuev D and Crogan M. *Discriminators for use in flow-based classification*. Queen Mary and Westfield College, Department of Computer Science, 2005.
36. sysdig; http://www.sysdig.org/ (2016, accessed 10 April 2016).
37. Yu L and Liu H. Feature selection for high-dimensional data: A fast correlation-based filter solution. In: *International conference on machine learning*, 2003, pp.856–863.
38. Collins RT, Liu Y and Leordeanu M. Online selection of discriminative tracking features. *Pattern Anal Mach Intell* 2005; 27(10): 1631–1643.
39. Bartlett G and Mirkovic J. Expressing different traffic models using the LegoTG framework. In: *Proceedings of the workshop on computer and networking experimental research using testbeds (CNERT)*. Piscataway, NJ: IEEE, 2015, pp.56–63.
40. Barbosa R, Sadre R, Pras A, et al. Simpleweb/university of twente traffic traces data repository. *Centre for Telematics and Information Technology, University of Twente*, 2010.
41. LBNL/ICSI enterprise tracing project. http://www.icir.org/enterprise-tracing (2015, accessed 27 July 2017).
42. Zou G, Kesidis G and Miller DJ. A flow classifier with tamper-resistant features and an evaluation of its portability to new domains. *Selected Areas in Communications* 2011.

43. Rossow C, Dietrich CJ, Grier C, et al. Prudent practices for designing malware experiments: Status quo and outlook. In: *Security and privacy* (SP). Piscataway, NJ: IEEE, 2012, pp.65–79.
44. Calvet J, Davis CR, Fernandez JM, et al. The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet. In: *Proceedings of the 26th annual computer security applications conference*. New York: ACM, 2010.
45. John JP, Moshchuk A, Gribble SD, et al. Studying spamming botnets using Botlab. In: *USENIX symposium on networked systems designs and implementation (NSDI)*, 2009. Berkeley, CA: Usenix.
46. Alwabel A, Shi H, Bartlett G, et al. Safe and automated live malware experimentation on public testbeds. In: *7th workshop on cyber security experimentation and test (CSET 14)*, 2014. Berkeley, CA: Usenix.
47. Lee CP. *Framework for botnet emulation and analysis*. PhD Thesis, Georgia Institute of Technology, 2009.
48. Jackson AW, Lapsley D, Jones C, et al. SLINGbot: A system for live investigation of next generation botnets. In: *Conference for homeland security, 2009. CATCH'09. Cybersecurity Applications & Technology*. Piscataway, NJ: IEEE, 2009, pp.313–318.

Author biographies

Z. Berkay Celik is a PhD student in the department of Computer Science at the Pennsylvania State University and a member of the Systems and Internet Infrastructure Security Laboratory (SIIS). His research interests include security, privacy, and machine learning.

Patrick McDaniel is a Distinguished Professor in the School of Electrical Engineering and Computer Science and Director of the Institute for Networking and Security Research at the Pennsylvania State University. Professor McDaniel is a Fellow of the IEEE and ACM and program manager and lead scientist for the Army Research Laboratory's Cyber-Security Collaborative Research Alliance. Patrick's research centrally focuses on a wide range of topics in security and technical public policy. Prior to joining Penn State in 2004, he was a senior research staff member at AT&T Labs-Research.

Thomas F. Bowen works on implementation of Cyber Virtual Ad hoc Network (CyberVAN) at Vencore Labs.