Towards Network Containment in Malware Analysis Systems

Mariano Graziano
Institut Eurecom
graziano@eurecom.fr

Corrado Leita
Symantec Research Labs
corrado_leita
@symantec.com

Davide Balzarotti
Institut Eurecom
balzarotti@eurecom.fr

ABSTRACT

This paper focuses on the containment and control of the network interaction generated by malware samples in dynamic analysis environments. A currently unsolved problem consists in the existing dependency between the execution of a malware sample and a number of external hosts (e.g. C&C servers). This dependency affects the repeatability of the analysis, since the state of these external hosts influences the malware execution but it is outside the control of the sandbox. This problem is also important from a containment point of view, because the network traffic generated by a malware sample is potentially of malicious nature and, therefore, it should not be allowed to reach external targets.

The approach proposed in this paper addresses the repeatability and the containment of malware execution by exploring the use of protocol learning techniques for the emulation of the external network environment required by malware samples. We show that protocol learning techniques, if properly used and configured, can be successfully used to handle the network interaction required by malware. We present our solution, Mozzie, and show its ability to autonomously learn the network interaction associated to recent malware samples without requiring a-priori knowledge of the protocol characteristics. Therefore, our system can be used for the contained and repeatable analysis of unknown samples that rely on custom protocols for their communication with external hosts.

Keywords

Malware containment, protocol learning, network traffic replay

1. INTRODUCTION

Dynamic analysis is a useful instrument for the characterization of the behavior of malware samples. The most popular approach to perform dynamic analysis consists in the deployment of *sandboxes*, i.e., instrumented environments in which a malware sample is run, and in which detailed information on the actions performed by the sample is logged. A variety of different approaches has been explored for the collection of host-based information in sandboxed environments. These range from the API hooking approaches adopted by CWSandbox [4] and the Cuckoo sandbox [6], as well as more elegant and less intrusive techniques such as the ones used by TTAnalyze and Anubis [8, 3]. The output of the sandbox analysis has proven to be of extremely useful to malware analysts, both to study the execution of a single sample and to identify behavioral commonalities among different samples [9].

However, the result of the execution of a malware sample in a sandbox is highly dependent on the sample interaction with other Internet hosts. This was clearly described by Rossow et al. in [24], where the authors analyzed the interaction between malware and Internet hosts during extended execution in a sandboxed environment. The study underlined the critical dependance on remote hosts for the download of additional malware components and for the C&C coordination.

The network traffic generated by a malware sample also raises obvious concerns with respect to the containment of the malicious activity. In fact, it is important to ensure that the correct execution of the sample does not cause any damage to other external hosts, for instance in the context of self-propagation attempts. However, these concerns go beyond the containment issue and relate directly to the quality of the analysis itself. For instance, Leita et al. [17] compared the output of the clustering algorithm presented in [9] with other information sources, such as static malware clustering techniques and honeypot data. This empirical study clearly showed how polymorphic variants of the same malware sample could be erroneously associated to different groups depending on the state of their C&C server at the moment in which the samples were executed.

In this paper we address two problems. The first is the poor repeatability of malware analysis experiments. Malware analysts often execute samples inside a sandbox, in order to observe and collect their malicious behavior. However, the exhibited behavior may depend on external factors, such as the commands received by a C&C server or the content of a given URL. For example, consider a classic scenario common to many security companies. Collected samples are automatically analyzed by a malware analysis system, and their behavior (e.g., filesystem operations, process creations, and modification to the Windows registry) is stored in a database. When a program requires a closer look, an analyst can run it again in a separate, better in-

strumented environment, for example by using a debugger or by collecting all the system calls. Unfortunately, it is often the case that these "secondary inspections" are performed several days, or even weeks after the samples were initially collected, with the risk of studying dead samples for which the required infrastructure is not available anymore. In fact, the remote machines contacted by malware are volatile by nature, often hosted on other compromised computers, or taken down by providers and law enforcement when the malicious activity is detected. Therefore, the malware infrastructure required to properly run the sample is normally available for only a limited amount of time.

The second problem we address in this paper is related to malware containment, i.e., to the ability of properly execute a given sample in an isolated environment, where it cannot cause any harm to the rest of the world. In general, full containment of a new, previously unknown, sample is impossible. However, we can identify two scenarios in which such result could be achieved: the execution of a polymorphic variation of an already analyzed malware, and the re-execution of a previously studied sample. In both cases, the sample (or a behaviorally equivalent variation of it) has already been analyzed by the system and, therefore, the problem of full containment can be reduced to the previous problem of repeatable execution. The idea is that if we can "mimic" the behavior of the network to match the one observed in a previous execution, we can obtain at the same time a repeatable experiment and a containment of the malware execution.

To mitigate these issues, in this paper we want to study to which extent it is possible to enrich the information collected during malware execution to make the experiments repeatable and achieve full network containment. In particular, we experiment with a protocol-agnostic technique [20] previously adopted to model the attack traffic in high-interaction honeypots [18, 16]. The idea consists in building a finite state machine (FSM) of the network activity generated by each malware sample. The extracted FSM can be stored alongside the other collected information, and can then be used to properly "simulate" all the required endpoints whenever the sample needs to be analyzed again in the future.

It is important to note that in this paper we only address *network* repeatability: Our goal is to ensure that the malware finds all the remote components it needs to properly execute. On the contrary, we do not address full process repeatability, i.e., the problem of forcing two executions of a malicious executable to behave exactly the same from an operating system point of view. Balzarotti et al. [7] studied the problem of full repeatability in the context of the detection of split personalities in malware. Their prototype works at the system-call level and has several technical limitations, making it hard to deploy on current malware sandboxes.

To summarize, the paper makes the following contributions:

- We discuss how protocol learning techniques can be used to model the traffic generated during the execution of malware samples. In particular, we describe the limitations of protocol-agnostic approaches and show that, if properly setup and configured, they can be used to successfully replay real malware conversations.
- We describe the implementation of Mozzie, a network containment system that can be easily applied to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic and achieve full containment for real malware samples.

Approach	Containment	Quality
Full Internet access	×	∼
Filter/redirect specific ports	∼	∼
Common service emulation	√	∼
Full isolation	√	×

Table 1: Network access strategies in dynamic analysis

The rest of this paper is structured as follows. In Section 2 we introduce the problem of network repeatability and containment and we discuss related work on the topic. In Section 3 we briefly describe the core ideas underneath the protocol learning algorithms adopted in this paper. In Section 4 we present our approach, the architecture of our system, and the details of our prototype implementatoin of Mozzie. We then present and discuss the results of our experiments in Section 5, and and the limitations of our approach in Section 6. Finally we conclude the paper in Section 7.

2. MALWARE ANALYSIS AND CONTAINMENT

Several different strategies have been proposed in the state of the art to address the problem of network containmnent and the quality of the dynamic analysis. In particular, the concept of quality refers to both the need to allow connectivity to external hosts (to expose the malware interesting behavior) and to the need to make the analysis process repeatable. Table 1 summarizes the previous work in four different categories.

Full Internet access. The most straightforward approach consists in providing the sandbox with full Internet access. A similar approach is however unacceptable from a containment standpoint: the malware sample is left free to propagate to victims, or to participate to other types of malicious activities (DoS, spam, ...). The quality of the analysis is also only partially acceptable: the sample is left free to interact with external hosts upon execution, but its behavior becomes dependent on the state of external hosts, leading to the problems underlined in [17].

Filter/redirect specific ports. The containment problem associated to Full Internet access is rarely discussed in Internet-connected sandboxes such as Anubis [3], CWSandbox [4] and others. From informal discussions with the maintainers, it appears to be common practice for the public deployment of these sandboxes to employ simple filtering or redirection rules, in which TCP ports commonly associated to malicious scans (e.g. port 139 and port 445) are either blocked or redirected towards honeypots. This partially solves the containment problem: SMB vulnerabilities are a very common propagation vector for self-propagating

malware, that can be easily prevented with such measures. However, this approach is not able to deal with other types of activity whose nature cannot be easily discerned from the TCP destination port. A similar attempt to perform containment through redirection was implemented also in the context of honeyfarms such as Potemkin [28] and GQ [12]. In such deployments, the authors had investigated the idea of reflecting outbound traffic generated by infected virtual instances of the honeyfarm towards other instances of the same honeyfarm. A similar approach proved to be valuable for the analysis of malware propagation strategies, but was not effective at dealing with other types of traffic such as C&C communication. In fact, redirecting a C&C connection attempt towards a generic honeyfarm virtual machine is not likely to generate meaningful results. Kreibich et al. [15] have recently improved GQ making it a real and versatile malware farm. They have addressed the containment problem with precise policies but their approach has not dealt with the repeatability issue.

Common service emulation. Sandboxes such as Norman Sandbox prevent the executed malware from connecting to the Internet, and provide instead generic service implementations for common protocols such as HTTP, FTP, SMTP, DNS and IRC. A similar approach was revisited and enhanced by Ionue et al. in [14], a two-pass malware analysis technique in which the malware sample is allowed to interact with a "miniature network" generated by an Internet emulator able to provide a variety of dummy services to the executed malware sample. All these approaches are however limited, and rely on a-priori knowledge of the communication protocols employed by the malware sample. Malware often uses variations of standard protocols, or completely ad-hoc communication protocols that cannot be handled through dummy services. Yoshioka et al. [30] have tried to tackle this problem by incrementally refining the containment rules according to the dynamic analysis results. While such an approach provided an elegant solution to the containment problem, it did not address the quality of the analysis and it did not attempt to remove dependencies between the malware behavior and the state of the external Internet hosts involved in the analysis.

Full isolation. Completely preventing the malware sample from interacting with Internent hosts ensures a perfect containment of its malicious activity. However, the complete inability to interact with C&C servers and repositories of additional components is likely to severely bias the outcomes of the dynamic analysis process.

Table 1 underlines a partial trade-off between the containment problem and that of ensuring the quality and repeatability of the analysis. On the one hand, running the malware sample in full emulation addresses all the containment concerns but, by barring the malware sample from communicating with the external hosts it depends on, it strongly biases the results of the dynamic analysis (i.e., the sample may only go as far as trying to connect to the hosts but without exposing any real malicious behavior). On the other hand, providing the sandbox with full Internet connectivity increases the analysis quality but it does not solve the repeatability problem, and it also raises important ethical and legal concerns.

This paper aims at addressing this problem by exploring the use of protocol learning techniques to automatically create network interaction models for the hosts the malware depends on upon execution (even in presence of custom and undocumented protocols), and using such models to provide the sandbox with an isolated, yet rich network environment.

3. PROTOCOL INFERENCE

A common problem when looking at the network interaction generated by a malware sample is associated to the interpretation of application-level protocols. Malware samples often propagate by exploiting vulnerabilities in poorly documented protocols (such as SMB), and rely on custom protocols for their coordination with the C&C servers [26, 25, 27]. The execution of the sample in isolation requires us to trick the malware sample in interacting with replicas of the real Internet hosts the malware interacts with (victims, C&C servers, ...), but without being able to assume a-priori knowledge of the application-level protocol implemented in such services.

This challenge is addressed in this paper by resorting to protocol learning techniques. Techniques such as ScriptGen [20, 19], RolePlayer [13], Discoverer [11] and Netzob [5] aim at partially reconstructing the protocol message syntax and, in some cases, the protocol Finite State Automata [19, 5] from network interactions. The inference is performed by looking at network traces generated by clients and servers while minimizing the number of assumptions on the protocol characteristics. Differently from other approaches such as [10, 22, 21, 29], that factor into the protocol analysis also host-based information obtained through execution monitoring or memory tainting, the above methods focus on the extraction of the protocol format solely from network traces and are therefore particularly suitable to our task, where we are unable to control the remote endpoints contacted by malware.

While any of the previouly mentioned tools would be suitable for this task, in this paper we focus on ScriptGen [20, 19] because of its limited amount of assumptions on the protocol characteristics and its support to the inference not only of single protocol messages, but also of their structure within the protocol flow. ScriptGen is an algorithm that generates an approximation of a protocol language by means of a "server-side" Finite State Machine whose scope corresponds to a specific TCP connection or UDP flow: the FSM root corresponds to the establishment of the connection, while any of the FSM leaves corresponds to the point in which the connection is successfully closed. In the FSM representation each transition is labelled with a regular expression matching a possible client request, while each state is labelled with the server answer to be sent back to the client when reaching that specific state.

ScriptGen performs this task through two subsequent processes, as illustrated in Figure 1:

1. **Semantic clustering.** Initially introduced in [19], the semantic clustering algorithm aims at grouping together protocol messages likely to be semantically similar. Any new conversation to be added to an ex-

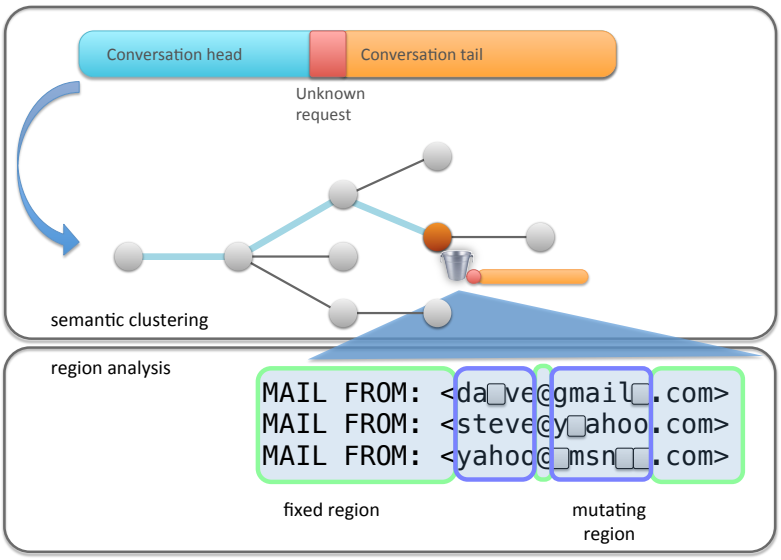

Figure 1: Simplified diagram of the ScriptGen operation

isting protocol model will consist of two parts: an initial part (*head*) composed of messages already modeled in the current version of the FSM, and a final part (*tail*) composed of messages that do not match the current model. A 0-length head represents a conversation whose first message already differs from the current protocol model, while a 0-length tail models a conversation that already fully matches the existing protocol knowledge. Semantic clustering aims at grouping together messages sharing the same head (by matching the head messages against the current FSM model) and likely to be sharing the same tail (by clustering conversations sharing the same head according to the length of the tail messages).

2. **Region analysis.** Semantic clustering leads to the identification of protocol messages that are very likely to be associated to a specific semantic context. The process that generalizes the specific protocol messages into a set of regular expressions able to correctly recognize future messages sharing the same semantic value is called region analysis and was introduced in [20]. Through the analysis of multiple versions of a semantically similar message, the region analysis algorithm aims at identifying *regions* with similar characteristics. Through the subsequent application of global alignment algorithms (Needleman-Wunsch [23]), ScriptGen refines the clustering of the messages separating those exposing major structural differences (macroclustering) while correctly dealing with variable length fields. The final result consists in a global alignment of each identified group. The statistical analysis of each byte of the alignment outcome is used to identify protocol *regions*, i.e., portions of protocol message likely to have a specific semantic value. The distribution of the content of each region over the different samples gives us information on their semantic nature: regions with random content throughout the set of samples are likely to be nonce values, regions with fixed content are likely to be separators or semantically reach protocol fields, and regions mutating through a limited amount of possible values are likely to be associated to more subtle protocol semantics, whose preservation is decided through a *microclustering* threshold. The final outcome of this semantic evaluation is a set of regular expressions, each of which will lead to the generation of a new subtree in the FSM object.

Most of the ScriptGen operation is driven by thresholds, that regulate for instance the different clustering steps. A simple approach for tuning the thresholds to the best configuration was introduced in [20] and consisted in brute-forcing all the possible combinations of thresholds to identify the global optimum. While this is an computationally expensive process, the computed thresholds proved to be sufficiently robust to handle protocols with "similar" characteristics in terms of amount of variability in their structure.

It is important to understand that ScriptGen avoids any assumption on the nature of the protocol separators or on the possible representation of semantically relevant fields, and performs a partial reconstruction of the protocol semantics by analyzing at the same time multiple samples of the same type of protocol exchange (*conversation*). The higher the number of conversations available to the algorithm, the more precise the protocol inference process will be. Intuitively, if we consider two message instances of a protocol containing a random cookie value, their cookie value could be *aabd* and *awed*. ScriptGen would have no way to consider such protocol section as a mutating region of 4 characters, since (by accident) both values start with 'a' and end with 'd'. These accidental false inferences can be filtered out only by considering a sufficiently large number of conversation samples.

ScriptGen has been successfully used in the past to automatically generate models of network behavior for the emulation of vulnerable services in exploits. This emulation was included in a distributed honeypot [16, 18] deployment, in order to emulate 0-day exploits and collect information

on the propagation vector of self-propagating malware. This paper investigates the use of ScriptGen in a different context, that of malware analysis, and tries to leverage its properties to automatically generate network models for the remote endpoints involved in the execution of a malware sample in a sandboxed environment.

4. SYSTEM OVERVIEW

Our approach to achieve repeatability and containment in malware analysis experiments can be summarized in four steps:

1. **Traffic Collection** - In this phase the system collects a number of network traces associated with the execution of a certain malware sample. This can be done by running the malware in a sandbox, or by extracting existing traces generated by past analyses.

2. **Endpoint Analysis** - This is a cleaning and normalization process applied to all the collected traces. Its main goal consists in removing anomalous traces that could affect the results and the associated conclusions. Each trace is then normalized to remove endpoint randomization, such as the one introduced by IP fluxing techniques.

3. **Traffic Modeling** - This phase aims at the automated generation of models starting from the collected traffic samples and at their subsequent storage in a compact representation. The modeling can be performed in two different ways: in an online fashion (from now on called *incremental learning*), in which the model is initially very simple and it is subsequently refined at every new execution of the sample, or in an *offline* fashion, more suitable for the analysis of previously collected network dumps. The actual logic used to model the traffic is implemented in a separate component of our system. As we already explained in Section 3, the current implementation is based on the ScriptGen approach, but other unsupervised algorithms can be easily plugged into our system to achieve the same result.

4. **Traffic Containment** - In this last phase, we use the model extracted in the previous step to mimic the network environment required by the malware sample. The containment system is implemented as a transparent proxy. When the model is sufficiently precise, the proxy is able to mimic the external world, effectively achieving "full containment". When the model is incomplete, the proxy redirects the requests it cannot handle to the real targets. In this case, the system also collects the forwarded traffic to improve the training set, effectively closing the loop back to Step 1.

In the rest of the section, we introduce each phase in details, and we describe how each of them have been implemented in our system.

4.1 Traffic Collection

Collecting the malware traffic is as simple as running a network sniffer while the sample is running in the sandbox. For instance, several online systems (e.g., Anubis [3], and CWSandbox [4]) allow users to download the pcap file recorded during the analysis phase.

As explained in Section 5, we used two different datasets for our experiments, one extracted from old Anubis reports, and one collected live by running the samples inside a Cuckoo's sandbox. Finally, in order to be consistent with the data collected in the past, also in our experiments we limited the malware analysis and the network collection time to five minutes per sample.

4.2 Endpoint Analysis

The second phase of our process consists in cleaning and normalizing the collected traffic to remove spurious traces and improve the effectiveness of the protocol learning phase when facing network-level randomization.

The cleaning phase mainly consists in grouping together traces that exhibit a comparable network behavior. The intuition underneath this cleaning process is that the traces may have been generated at different points in time, and may capture different "states" of the remote endpoints. Most of the traces are likely to have been generated when the malware was indeed fully active, but we may still have to deal with a minority of traces that may have been generated, for example, when the C&C server was temporarily not reachable. It should be noted that the semantic clustering process explained in Section 3 allows ScriptGen to correctly deal with these cases. However, in this paper we are interested in evaluating the efficiency of our method at correctly leveraging *useful* traces to generate usable models, and thus we choose to clean the dataset from these spurious traces. In practice, this is achieved by clustering together traces according to each involved destination endpoint, where endpoint is defined as an $(IP, port)$ tuple. We consider the cluster with the highest amount of traces having similar high-level network behavior as the one representing the state of interest for our experiments.

While the cleaning process succeeds in most of the cases, in some of our experiments we noted that the clustering algorithm failed to identify a predominant network behavior for a specific sample. Closer investigation revealed that this failure is associated to the introduction of randomization in the network behavior of the sample. The most common example of this phenomenon is associated to malware using IP fluxing techniques. IP flux, also known as fast-flux, is a DNS-based technology used by malware writers to improve the resilience of their (often botnet) architecture. The idea is to rapidly swap the IP address associated to a particular domain name, to avoid having a single-point of failure and reducing the effect of IP blacklisting.

When a malware uses IP fluxing, most of the collected network traces will involve different endpoints. In other words, instead of having ten samples of conversations associated to a single endpoint, we will have ten different targets associated to one conversatione each. As we will see in the discussion of the Traffic Modeling component, this situation plays against our choice of creating protocol models on a per-endpoint basis: each endpoint will not be associated to a sufficient amount of samples to generate a meaningful model. However, this phenomenon is easy to detect because our endpoint clustering component would return an unusual result composed of many clusters containing a single trace each. In this case we automatically "normalize" the endpoints by identifying the DNS request that returned different IP addresses and by forcing it to return always the same value. In these cases, our system automatically replaces each

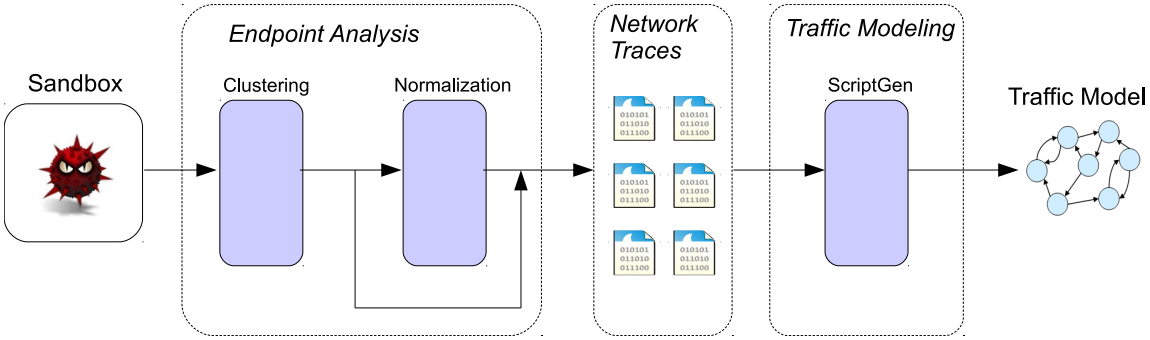

Figure 2: Creation of a Traffic Model

fluxed IP with the normalized IP in all the subsequent network flows. By performing this simple step we can obtain an uniform learning dataset, ready to be analyzed by the next stage of our system.

4.3 Traffic Modeling

Starting from the set of network traces obtained from the previous phase, the Traffic Modeling phase leverages the protocol learning algorithm (in our case, ScriptGen) to generate models for the network interaction with each endpoint involved in the malware execution. The models are maintained in the form of a dictionary, where each encoutered destination IP address and destination port is associated to its corresponding model (see Figure 2). This dictionary is later used to mimic the whole network environment, with the goal of containing all the requests generated by the malware.

4.4 Containment Phase

The goal of the containment phase is to "trick" the malware sample into believing that it is connected to the Internet, while in fact all the traffic is artificially generated by leveraging the information contained in the current protocol model.

The main component of the containment phase is the FSM player. The player is responsible for analyzing the incoming packets looking for new connections attempts, and checking if the contacted endpoint is present in the dictionary of FSMs. If so, the corresponding model is loaded and associated to the connection. Then, for each message, the player analyzes the current state and computes the next state. This is done by finding the best transition that matches the incoming content and by extracting the corresponding list of possible answers.

Figure 3 shows the operation of our system over two possible operation modes: *full* and *partial containment*.

In the *full containment* case, the protocol model is accurate enough to allow our system to correctly generate a response for every network interaction generated by the malware upon execution.

In the *partial containment* case, instead, the protocol model is inaccurate or incomplete: some of the network interactions created by the malware are not modelled in the associated FSM. Whenever a message cannot be matched with the current FSM, the system is unable to further emulate the associated remote endpoint. In such case, the system enters in replay mode for that specific endpoint, and replays all the traffic generated so far towards the real Internet host associated to it. By replaying all the traffic generated so far, we are able to handle possible authentication steps that the malware may have already performed in its interaction with the FSM. The answer to the unknown request is eventually delivered by the real endpoint, and all the subsequent interaction is then relayed by the system (proxy mode).

This process needs to be carried out without affecting the malware execution. For instance, consider the example in Figure 4 in which the malware sends three messages $\{Msg_1, Msg_2, Msg_3\}$ over a TCP connection. The player is able to follow the first two messages on the FSM, thus returning the corresponding responses $\{Resp_1, Resp_2\}$. However, the third message is different from what it was expecting, and it does not know what to answer. Therefore, Mozzie opens a new TCP connection to the original target, and quickly replay the messages 1 and 2 in chronological order to bring the new connection to the same state of the one it is simulating with the malware. Then it sends the Msg_3 and switches to proxy mode. In proxy mode, the system acts like a transparent proxy, forwarding each packet back and forth from the malware to the endpoint on the Internet. When the connection is terminated, the data is used to incrementally improve the model, so that it could handle the same conversation in the future.

By executing a malware sample multiple times, we are therefore able to gradually and automatically move from partial containment (in case in which part of the generated interacton is still unknown) to full containment, where all the malware network behavior is fully modeled and emulated by the system. However, it should be noted that the full learning of the network behavior of the malware is different from the full knowledge of the protocol. It is possible to follow a malware without contacting the external server in all its executions within the sandbox, but this does not mean all the commands of the protocols have been discovered.

4.5 System Implementation

In the previous sections we explained the architecture of the system. Now we can describe how Mozzie, our prototype implementation, is realized.

Mozzie is based on *iptables*. In particular it uses the NFQUEUE userspace packet handler with the Python nfqueue-bindings [2]. This allows our user-space component to accept, drop, or modify each incoming packet. To decode the packets that are in the queue, we used the Scapy [1] library.

Our current prototype handles three different IP proto-

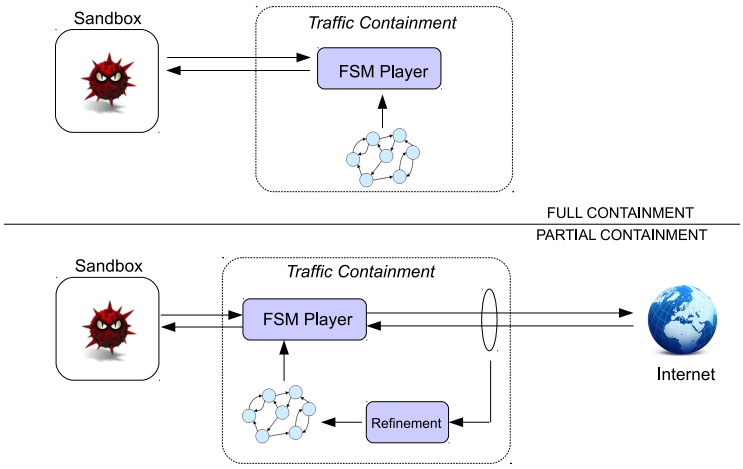

Figure 3: Replaying Architecture

cols: ICMP, UDP and TCP. ScriptGen cannot model the ICMP protocol, because it lacks the concept of *port* that is required to build the Finite State Machine. Therefore, Mozzie intercepts all the ICMP ECHO request messages and always answers with an ECHO reply. Beside that, the system mainly works as a userspace NAT, changing the destination IP and port for each TCP or UDP packet, to redirect them to the emulator responsible for that endpoint. The emulator runs an implementation of the ScriptGen algorithm and one instance of the FSM Player for each endpoint contacted by the malware. For example, if the malware opens a new TCP connection toward $(IP, Port)$, Mozzie checks if a FSM exists for that endpoint. If it finds one, it starts one FSM Player process to handle the connection and start redirecting the packets toward it. If not, it let the packets pass through so they can reach the real destination on the Internet.

Finally, the endpoint analysis is implemented as a series of Python scripts. One is responsible to process the available network traces and to cluster them together according to the contacted endpoints. The normalization is implemented by a separate tool that dissects the packets, changes the answer of the DNS requests, and replaces the corresponding IPs in the rest of the network traffic.

5. EVALUATION

In this section we describe the experiments we performed to evaluate Mozzie's ability to model real malware traffic. All the experiments were performed on an Ubuntu 10.10 machine running ScriptGen, Mozzie, and iptables v1.4.4. To perform the live experiments, we ran all samples in a Cuckoo Sandbox [6] running a Windows XP SP3 virtual machine.

5.1 System Setup

The goal of our evaluation is to automatically find the minimum number of network traces required to generate a finite state machine that can be used to fully contain the network traffic generated by a given malware sample.

To reach this goal, the first step of our experiments consisted in testing ScriptGen and properly tuning its parameters for the protocols we wanted to model. In fact, in our system we use ScriptGen to model the network behavior of

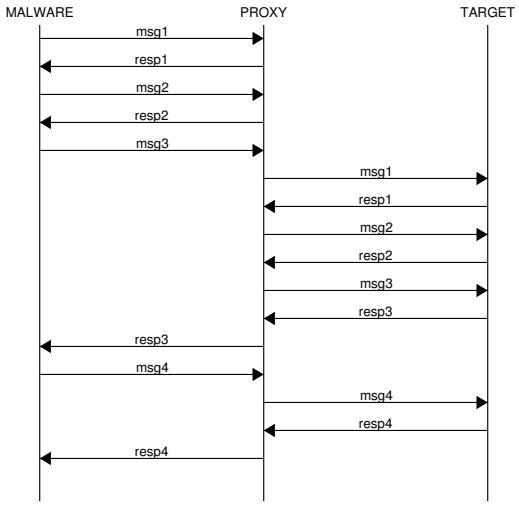

Figure 4: Sequence of messages during traffic replay

a generic program, but this is not the scenario for which the learning protocol tool was designed in the first place. As we already explained in Section 3, the best thresholds of ScriptGen's learning algorithm were experimentally set to the values that were observed to work well (in average) for network worms and remote exploits. However, those thresholds need to be re-computed for different protocols, in particular when moving from a text-based (e.g., HTTP) to a binary format (e.g., RPC).

In the first part of our experiments, we performed a number of tests to learn the optimal parameters for a number of protocols that are commonly used by several malware samples, namely HTTP, IRC, DNS, and SMTP. This step requires the algorithm to be executed several times on the same protocol traces, each time with different parameters [1]. Once the optimal setup was reached, we reused the same values for all the malware samples that used the same protocol. However, in one of the experiments, we discovered that the

[1]Please refer to [20, 19] for a description of the procedure required to set the thresholds

malware under analysis implemented a custom binary protocol. Since we did not have the ScriptGen configuration for that protocol, we had to re-apply the learning phase for that particular sample traffic.

Even though this operation can take several hours, it is important to note that if the thresholds are not set to their optimal values our technique would still be able to model unknown protocols, even though the system would require an higher number of traces to reach the full containment.

5.2 Experiments

Our experiments with real malware can be divided in two groups. In the first case, that we label *"offline"* learning, we use our system to model old traces collected in the past for polymorphic samples. Malware sandboxes normally avoid executing the same sample multiple times, returning the previously computed results when they recognize (usually from the MD5) that a file was already analyzed in the past. However, in case of polymorphic variations, it is possible that the same malware family gets executed several times. Based on this observation, we extracted from the Anubis [3] database the network traffic dumps associated to five polymorphic samples that adopted cleartext protocols in their communication.

Our goal was to show that, by using these traces, we can model the network behavior of the malware, and use the extracted FSM to replay and contain the execution of any other polymorphic variation of the same sample.

The results of this first experiment are reported in Table 2. The first two columns report the antivirus label and the malware category associated to each sample. The next column reports the success of the experiment, where a `FULL` (100%) value means that full containment was achieved and all the packets were properly replayed. In only one case, for the Koobface malware, our system was not able to successfully model the entire network traffic. The reason is the fast flux approach adopted by Koobface in which both the domain names *and* the IP addresses rotate. This makes it impossible for Mozzie to correctly model the DNS protocol, since there are not two sequences of request/response that look the same in the dataset.

Column 4 shows whether the normalization step was applied to the traffic. As we already explained in Section 4, clustering is required to deal with noise in the network traces, most of the time introduced by an anomalous execution of the malware (e.g., due to a network timeout on a web request). On top of that, certain malwares require a normalization phase to properly sanitize the traces from randomization introduced by IP or domain flux techniques.

Finally, the last column of the table shows the number of input traces required to successfully model the traffic. We started the experiment by running Mozzie on a single network trace. We then loaded the extracted model in a virtual machine and used it to try to contain five consecutive executions of another polymorphic variation of the same malware. The reason behind the five runs is that we wanted to be sure that ScriptGen did not return the right message by chance, and that the experiment can be reliably repeated multiple times. If the extracted model was not sufficient to properly "replay" the network conversation, we added one more network trace to the learning pool and repeated the experiment. The number in the fifth column represents the number of network traces required to create a Finite State Machine that achieved full containment (or its best approximation in the case of Koobface) of the malware sample. The results vary between 9 and 23 traces. These numbers may seem large, but it is important to remember that ScriptGen is completely protocol agnostic and that each experiment was performed starting with an empty protocol model. For example, we discovered that 6 traces is the minimum amount required to properly model a DNS request/response exchange (due to the fact that the response has to contain the same request ID field used in the request).

Table 3 reports the result of our second group of tests. In this second experiment, we focused on incremental learning, i.e., on analyzing current malware in a sandbox environment each time refining our model of the network traffic. We started by executing the samples 3 times, without attempting to replay the traffic. Then we created our first model, and started executing the sample in the sandbox with Mozzie acting as a proxy. Whenever the system was not able to contain the traffic, the requests were forwarded to the real servers, and the FSM updated with the new information. The third column reports the number of time each malware has to be analyzed before the model can achieve full containment for five consequent runs.

Overall, we tested 2 IRC botnets, 1 HTTP botnet, 4 droppers, 1 ransomware, 1 backdoor and 1 keylogger. For these samples, we needed a number of network traces ranging from 4 to 25. The first number seems in contradiction with the lower bound we have previously found. The truth is that this particular sample does not use DNS and thus contact the C&C servers directly by using an hardcoded IP address. For all the other malware that generate DNS traffic the number is definitively higher than the lower bound. On average we need 14 traces to be able to build a good traffic model.

Certainly, large malware analysis systems forced to analyze tens or hundreds of thousand of samples per day cannot afford to repeat the tests 14 times. However, such a high number of new samples collected every day is largely due to the common use of polymorphism and packing techniques by malware writers. Therefore, once a FSM is available for one of the samples in the family, any further variation that preserves the behavior of the program does not require any additional training. Our system could help analyzing polymorphic samples for which the required network infrastructure is not available any more, and that nowadays cannot be tested at all. This, as we already described in the introduction, can improve the result of clustering, and can help malware analysts to properly label those samples that do not work anymore at the time of the analysis. Even better, our system could be used to replicate a specific network scenario that is targeted by a malware infection. Recent years have seen the rise of sophisticated attacks targeting specific environments, such as Stuxnet and Duqu [26, 25]. In these cases, the network traces obtained from the targeted network infrastructure (e.g. traces of interaction in a DCS system in a power generation control system) could be used to build a model of the targeted network environment, allowing the analysis of the malware to be successfully performed inside traditional, and safe, sandboxes.

6. LIMITATIONS

The current prototype of our containment system has several limitations. Some are specific to the way the system has

Sample	Category	Containment	Endpoint Normalization	Traces
W32/Virut	IRC Botnet	FULL	NO	15
PHP/Pbot.AN	IRC Botnet	FULL	NO	12
W32/Koobface.EXT	HTTP Botnet	72%	YES	9
W32/Agent.VCRE	Dropper	FULL	NO	23
W32/Agent.XIMX	Dropper	FULL	YES	10

Table 2: Results of the Offline learning Experiments

Sample	Category	Runs	Containment	Endpoint Normalization
W32/Banload.BFHV	Dropper	23	FULL	NO
W32/Downloader	Dropper	25	FULL	NO
W32/Troj_Generic.AUULE	Ransomware	4	FULL	NO
W32/Obfuscated.X!genr	Backdoor	6	FULL	NO
SCKeylog.ANMB	Keylogger	14	FULL	YES

Table 3: Results of the Incremental learning Experiments

been implemented and some are related to the self-imposed constraints associated to the chosen methodology.

More specifically, we can group the current limitations into three main families:

- The method adopted in this paper is completely protocol-agnostic. While its nature allows us to guarantee our ability to handle custom, undocumented protocols that can be adopted by future malware, it also imposes unnecessary constraints when dealing with simpler and well known protocols such as DNS. We have already seen that our system requires six samples of network interaction to learn how to properly replay a DNS request. The same result could be easily achieved by analyzing only one request, parsing the DNS fields, and extracting the required information in an ad-hoc fashion. However, the goal of this paper was to show how far it is possible to go with a completely generic system. Therefore, our results can be considered as a upper bound, as the system could be easily improved by adding ad-hoc handlers for common and well known protocol interactions.

- Our current prototype is implemented as a network proxy. Even though this approach has some advantages (e.g., it can be easily plugged into any existing sandbox), it makes the analysis of encrypted protocols impossible. However, this is mostly a technical limitation. The same approach could be implemented at the API level, where most of the network traffic is still available in clear text. Most of the malware sandbox environments already hook into the Windows API to extract information about the malware behavior. By adding our system to the hooked network and cryptographic APIs, we could intercept the communication on the host side and achieve full containment also for some encrypted protocols (e.g., the ones based on SSL).

- Our approach is very inefficient when a malware sample exhibits different behaviors independently of the input it receives from the network. For example, if a sample randomly selects the action to perform out of many possible options, Mozzie would require a lot of traces to properly model all possible behaviors. As an extreme case, domain flux techniques (or large pools of domain names like the one described in Section 5) cannot be modeled by our system without requiring protocol-aware heuristics, such as handling the DNS interaction by using a custom DNS service.

7. CONCLUSIONS

This paper addresses the problem of network containment and repeatability in the context of dynamic analysis tools such as sandboxes. As pointed out by previous work [17, 24] malware execution behavior has often a strong dependency with the state and the behavior of external hosts This raises repeatability issues: a malware analysis cannot be reproduced if the state of these external hosts has changed. At the same time, containment concerns are always associated to the communication of malware with external hosts, concerns that have only partially been addressed by the current state of the art.

We have discussed how protocol learning, used in the previous works in the context of service emulation for server-side honeypots, can be successfully used in this new context to provide an emulated, and contained network environment that allows correct execution of malware samples even in presence of undocumented, ad-hoc communication protocols.

We have described the implementation of Mozzie, a network containment system that can be easily adapted to all the existing sandbox environments. According to our experiments, an average of 14 network traces are required by Mozzie to model the traffic by approaching the problem of sandbox network emulation in a completely generic, protocol-agnostic way that can be applied to real-world malware samples.

The benefits of the large-scale application of similar techniques are significant: old malware samples whose C&C infrastructure has been shut down can be analyzed in the same network conditions they were supposed to find when active; in-depth analyses of samples of interest can be carried out in complete isolation, e.g. without direct connectivity to the underlying C&C server and thus without disclosing details on the analysis operation to the bot herders; malware targeting specific network environments (e.g. industrial control systems) can be analyzed in a replica of the network layout they expect to find.

Acknowledgements

This work has been partially supported by the European Commission Seventh Framework Programme (FP7/2007-2013) under grant agreement 257007 and through the FP7-SEC-285477-CRISALIS project.

8. REFERENCES

[1] Scapy. `http://www.secdev.org/projects/scapy/`, 2003.

[2] nfqueue-bindings. `://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki/`, 2008.

[3] Anubis. `http://anubis.iseclab.org`, 2009.

[4] Cwsandbox. `http://www.mwanalysis.org`, 2009.

[5] Netzob. `http://www.netzob.org`, 2009.

[6] Cuckoo Sandbox. `http://www.cuckoosandbox.org`, 2010.

[7] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2010.

[8] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[9] U. Bayer, P. Milani Comparetti, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symp. on Network and Distributed System Security (NDSS)*, 2009.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *14th ACM conference on Computer and Communications Security*, pages 317–329. ACM New York, NY, USA, 2007.

[11] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *16th USENIX Security Symposium*, 2007.

[12] W. Cui, V. Paxson, and N. Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical report, ICSI Tech Report TR-06-004, September 2006.

[13] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *The 13th Annual Network and Distributed System Security Symposium (NDSS)*, February 2006.

[14] D. Inoue, K. Yoshioka, M. Eto, Y. Hoshizawa, and K. Nakao. Malware behavior analysis in isolated miniature network for revealing malware‘s network activity. In *Proceedings of IEEE International Conference on Communications, ICC 2008, Beijing, China, 19-23 May 2008*, pages 1715–1721. IEEE, 2008.

[15] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. Gq: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011.

[16] C. Leita. *SGNET: automated protocol learning for the observation of malicious threats*. PhD thesis, University of Nice-Sophia Antipolis, December 2008.

[17] C. Leita, U. Bayer, and E. Kirda. Exploiting diverse observation perspectives to get insights on the malware landscape. In *DSN 2010, 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010.

[18] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *7th European Dependable Computing Conference (EDCC 2008)*, May 2008.

[19] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2006.

[20] C. Leita, K. Mermoud, and M. Dacier. Scriptgen: an automated script generation tool for honeyd. In *21st Annual Computer Security Applications Conference*, December 2005.

[21] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *15th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[22] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Atlanta, GA, USA, November 2008.

[23] S. Needleman and C. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*. J Mol Biol. 48(3):443-53, 1970.

[24] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. C. Freiling, and N. Pohlmann. Sandnet: Network Traffic Analysis of Malicious Software. In *1st Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, April 2011.

[25] Symantec. The Stuxnet worm. `http://go.symantec.com/stuxnet`.

[26] Symantec. W32.Duqu, the precursor to the next Stuxnet. http://go.symantec.com/duqu.

[27] Symantec. W32.Koobface. `http://www.symantec.com/security_response/writeup.jsp?docid=2008-080315-0217-99`.

[28] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review*, 39(5):148–162, 2005.

[29] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[30] K. Yoshioka, T. Kasama, and T. Matsumoto. Sandbox analysis with controlled internet connection for observing temporal changes of malware behavior. In *2009 Joint Workshop on Information Security (JWIS 2009)*, 2009.