JACKSTRAWS: Picking Command and Control Connections from Bot Traffic

Gregoire Jacob
University of California, Santa Barbara
gregoire.jacob@gmail.com

Ralf Hund
Ruhr-University Bochum
ralf.hund@rub.de

Christopher Kruegel
University of California, Santa Barbara
chris@cs.ucsb.edu

Thorsten Holz
Ruhr-University Bochum
thorsten.holz@rub.de

Abstract

A distinguishing characteristic of bots is their ability to establish a command and control (C&C) channel. The typical approach to build detection models for C&C traffic and to identify C&C endpoints (IP addresses and domains of C&C servers) is to execute a bot in a controlled environment and monitor its outgoing network connections. Using the bot traffic, one can then craft signatures that match C&C connections or blacklist the IP addresses or domains that the packets are sent to. Unfortunately, this process is not as easy as it seems. For example, bots often open a large number of additional connections to legitimate sites (to perform click fraud or query for the current time), and bots can deliberately produce "noise" – bogus connections that make the analysis more difficult. Thus, before one can build a model for C&C traffic or blacklist IP addresses and domains, one first has to pick the C&C connections among all the network traffic that a bot produces.

In this paper, we present JACKSTRAWS, a system that accurately identifies C&C connections. To this end, we leverage host-based information that provides insights into which data is sent over each network connection as well as the ways in which a bot processes the information that it receives. More precisely, we associate with each network connection a behavior graph that captures the system calls that lead to this connection, as well as the system calls that operate on data that is returned. By using machine learning techniques and a training set of graphs that are associated with known C&C connections, we automatically extract *and* generalize graph templates that capture the core of different types of C&C activity. Later, we use these C&C templates to match against behavior graphs produced by other bots. Our results show that JACKSTRAWS can accurately detect C&C connections, even for novel bot families that were not used for template generation.

1 Introduction

Malware is a significant threat and root cause for many security problems on the Internet, such as spam, distributed denial of service attacks, data theft, or click fraud. Arguably the most common type of malware today are *bots*. Compared to other types of malware, the distinguishing characteristic of bots is their ability to establish a *command and control (C&C) channel* that allows an attacker to remotely control and update a compromised machine. A number of bot-infected machines that are combined under the control of a single entity (called the *botmaster*) are referred to as a *botnet* [7, 8, 14, 37].

Researchers and security vendors have proposed many different host-based or network-based techniques to detect and mitigate botnets. Host-based detectors treat bots like any other type of malware. These systems (e.g., anti-virus tools) use signatures to scan programs for the presence of well-known, malicious patterns [43], or they monitor operating system processes for suspicious activity [26]. Unfortunately, current tools suffer from low detection rates [4], and they often incur a non-negligible performance penalty on end users' machines. To complement host-based techniques, researchers have explored network-based detection approaches [15–18, 34, 41, 45, 49]. Leveraging the insight that bots need to communicate with their command and control infrastructure, most network-based botnet detectors focus on identifying C&C communications.

Initially, models that match command and control traffic were built manually [15, 17]. To improve and accelerate this slow and tedious process, researchers proposed automated model (signature) generation techniques [34, 45]. These techniques share a similar work flow (a work flow that, interestingly, was already used in previous systems to extract signatures for spreading worms [25,27,29,31,39]): First, one has to collect traces of malicious traffic, typically by running bot samples in a controlled environment. Second, these traces are checked for strings (or token sequences) that appear frequently, and can thus be transformed into signatures.

While previous systems have demonstrated some success with the automated generation of C&C detectors based on malicious network traces, they suffer from

three significant shortcomings: The first problem is that bots do not only connect to their C&C infrastructure, but frequently open many additional connections. Some of the additional connections are used to carry out malicious activity (e.g., scanning potential victims, sending spam, or click fraud). However, in other cases, the traffic is not malicious *per se*. For example, consider a bot that connects to a popular site to check the Internet connectivity, or a bot that attempts to obtain the current time or its external IP address (e.g., local system settings are under the control of researchers who might try to trick malware and trigger certain behaviors; they are thus unreliable from the bot perspective [19, 35]). In most of these cases, the malware traffic is basically identical to traffic produced by a legitimate client. Of course, one can use simple rules to discard some of the traffic (scans, spam), but other connections are much harder to filter; e.g., how to distinguish a HTTP-based C&C request from a request for an item on a web site? Thus, there is a significant risk that automated systems produce models that capture legitimate traffic. Unfortunately, a filtering step can remove such models only to a certain extent.

To highlight the difficulty of finding C&C connections in bot traffic, we report on the analysis of a database that was given to us by a security company. This database contains network traffic produced by malware samples run in a dynamic analysis environment. Over a period of two months (Sept./Oct. 2010), this company analyzed 153,991 malware samples that produced a total of 593,012 connections, *after* removing all empty and scan-related traffic. A significant majority (87.9%) of this traffic was HTTP, followed by mail traffic (3.8%) and small amounts of a wide variety of other protocols (including IRC). The company used two sets of signatures to analyze their traffic: One set matches known C&C traffic, the other set matches traffic that is known to be harmless. This second set is used to quickly discard from further analysis connections that are known to be unrelated to any C&C activity. Such connections include accesses to ad networks, search engines, or games sites. Using these two signature sets, we found 109,600 malicious C&C connections (18.5%), but also 69,211 benign connections (11.7%). The remaining 414,201 connections (69.8%) were unknown; they did not match any signature, and thus, likely consist of a mix of malicious and harmless traffic. This demonstrates that it is challenging to distinguish between harmless web requests and HTTP-based C&C connections.

The second problem with existing techniques is that attackers can confuse automated model (signature) generation systems: previous research has presented "noise injection" attacks in which a malware crafts additional connections with the sole purpose to thwart signature extraction techniques [10, 11, 33]. A real-world example for such a behavior can be found in the *Pushdo* malware family, where bots, in certain versions, create junk SSL connections to more than 300 different web sites to blend in with benign traffic [1].

The third problem is that existing techniques do not work when the C&C traffic is encrypted. Clearly, it is not possible to extract a content signature to model encrypted traffic. However, even when the traffic is encrypted, it would be desirable to add the C&C server destinations to a blacklist or to model alternative network properties that are not content-based. For this, it is necessary to identify those encrypted malware connections that go to the C&C infrastructure and distinguish them from unrelated but possibly encrypted traffic, such as legitimate, SSL-encrypted web traffic.

The root cause for the three shortcomings is that existing approaches extract models *directly* from network traces. Moreover, they do so at a purely syntactic level. That is, model generation systems simply select elements that occur frequently in the analyzed network traffic. Unfortunately, they lack "understanding" of the purpose of different network connections. As a result, such systems often generate models that match irrelevant, non-C&C traffic, and they incorrectly consider decoy connections. Moreover, in the case of encrypted traffic, no frequent element can be found at all.

To solve the aforementioned problems, we propose an approach to detect the network connections that a malware program uses for command and control, and to distinguish these connections from other, unrelated traffic. This allows us to immediately consider the destination hosts/domains for inclusion in a blacklist, even when the corresponding connections are encrypted. Moreover, we can feed signature generation systems with *only* C&C traffic, discarding irrelevant connections and making it much more difficult for the attacker to inject noise.

We leverage the key observation that we can use host-based information to learn more about the semantics of network connections. More precisely, we monitor the execution of a malware process while it communicates over the network. This allows us to determine, for each request, which data is sent over the network and where this data comes from. Moreover, we can determine how the program uses data that it receives over the network. Using this information, we can build models that capture the host-based activity associated with individual network connections. Our models are behavior graphs, where the nodes are system calls and the edges represent data flows between system calls.

We use machine-learning to build graph-based models that characterize malicious C&C connections (e.g., connections that download binary updates that the malware later executes, or connections in which the malware uploads stolen data to a C&C server). More precisely, start-

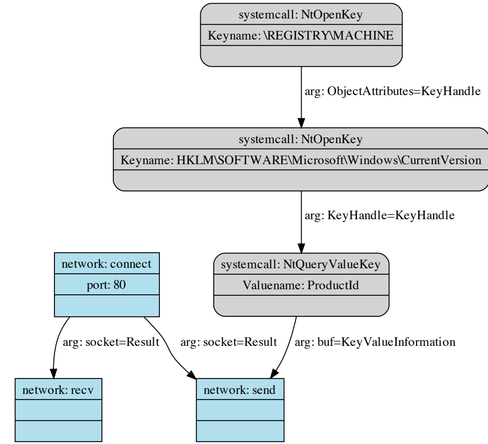
ing from labeled sets of graphs that are related to both known C&C connections and other, irrelevant malware traffic, we identify those subgraphs that are most characteristic of C&C communication. In the next step, we abstract from these specific subgraphs and produce *generalized graph templates*. Each graph template captures the core characteristics of a different type or implementation of C&C communication. These graph templates can be used to recognize C&C connections of bots that have not been analyzed previously. Moreover, our templates possess explanatory capabilities and can help analysts to understand how a particular bot utilizes its C&C channel (e.g., for binary updates, configuration files, or information leakage).

Our experiments demonstrate that our system can generate C&C templates that recognize host-based activity associated with known, malicious traffic with high accuracy and very few false positives. Moreover, we show that our templates also generalize; that is, they detect C&C connections that were previously unknown.
The contributions of this paper are the following:

- We present a novel approach to identify C&C communication in the large pool of network connections that modern bots open. Our approach leverages host-based information and associates models, which are based on system call graphs, with the data that is exchanged over network connections.
- We present a novel technique that generalizes system call graphs to capture the "essence" of, or the core activities related to, C&C communication. This generalization step extends previous work on system call graphs, and provides interesting insights into the purpose of C&C traffic.
- We implemented these techniques in a tool called JACKSTRAWS and evaluated it on 130,635 connections produced by more than 37 thousands malware samples. Our results show that the generated templates detect known C&C traffic with high accuracy, and less than 0.2% false positives over harmless traffic. Moreover, we found 9,464 previously-unknown C&C connections, improving the coverage of hand-crafted network signatures by 60%.

2 System Overview

Our system monitors the execution of a malware program in a dynamic malware analysis environment (such as Anubis [20], BitBlaze [40], CWSandbox [44], or Ether [9]). The goal is to identify those network connections that are used for C&C communication. To this end, we record the activities (in our case, system calls) on the host that are related to data that is sent over and received through each network connection. These activities are modeled as *behavior graphs*, which are graphs that capture system call invocations and data flows be-




```
GET /bot/doit.php?v=3&id=ec32632b-29981-349-398...
```

Figure 1: Example of behavior graph that shows information leakage. Underneath, the network log shows that the Windows ID was leaked via the GET parameter `id`.

tween system calls. In our setting, one graph is associated with each connection. As the next step, all behavior graphs that are created during the execution of a malware sample are matched against templates that represent different types of C&C communication. When a graph matches a template sufficiently closely, the corresponding connection is reported as C&C channel.

In the following paragraphs, we first discuss behavior graphs. We then provide an overview of the necessary steps to generate the C&C templates.

Behavior graphs. A behavior graph G is a graph where nodes represent system calls. A directed edge e is introduced from node x to node y when the system call associated with y uses as argument some output that is produced by system call x. That is, an edge represents a data dependency between system calls x and y. Behavior graphs have been introduced in previous work as a suitable mechanism to model the host-based activity of (malware) programs [5,13,26]. The reason is that system calls capture the interactions of a program with its environment (e.g., the operating system or the network), and data flows represent a natural dependence and ordered relationship between two system calls where the output of one call is directly used as the input to the other one.

Figure 1 shows an example of a behavior graph. This graph captures the host-based activity of a bot that reads the Windows serial number (ID) from the registry and sends it to its command and control server. Frequently, bots collect a wealth of information about the infected, local system, and they send this information to their C&C servers. The graph shows the system calls that are invoked to open and read the Windows ID key from the registry. Then, the key is sent over a network connec-

tion (that was previously opened with `connect`). An answer is finally received from the server (`recv` node).

While behavior graphs are not novel *per se*, we use them in a different context to solve a novel problem. In previous work, behavior graphs were used to distinguish between malicious and benign program executions. In this work, we link behavior graphs to network traffic and combine these two views. That is, we use these graphs to identify command and control communication amidst all connections that are produced by a malware sample.

C&C templates. As mentioned previously, the behavior graphs that are produced by our dynamic malware analysis system are matched against a set of C&C templates. C&C templates share many similarities with behavior graphs. In particular, nodes n carry information about system call names and arguments encoded as labels l_n, and edges e represent data dependencies where the type of flow is encoded as labels l_e. The main difference to behavior graphs is that the nodes of templates are divided into two classes; core and optional nodes. Core nodes capture the necessary parts of a malicious activity, while optional nodes are only sometimes present.

To match a C&C template against a behavior graph G, we define a similarity function δ. This function takes as input the behavior graph G and a C&C template T and produces a score that indicates how well G matches the template. All core nodes of a template must at least be present in G in order to declare a match.

Template generation. Each C&C template represents a certain type of command and control activity. We use the following four steps to generate C&C templates:

In the **first step**, we run malware executables in our dynamic malware analysis environment, and extract the behavior graphs for their network connections. These connections can be benign or related to C&C traffic.

JACKSTRAWS requires that some of these connections are labeled as either malicious or benign (for training). In our current system, we apply a set of signatures to all connections to find *(i)* known C&C communication and *(ii)* traffic that is known to be *unrelated* to C&C. Note that we have signatures that explicitly identify benign connections as such. The signatures were manually constructed, and they were given to us by a network security company. By matching the signatures against the network traffic, we find a set of behavior graphs that are associated with known C&C connections (called *malicious graph set*) and a set of behavior graphs associated with non-C&C traffic (called *benign graph set*). These sets serve as the basis for the subsequent steps.

It is important to observe that our general approach only requires labeled connections, without considering the payload of network connections. Thus, we could use other means to generate the two graph sets. For example, we can add a graph to the malicious set if the network connection corresponding to this graph contacted a known blacklisted C&C domain. This allows us to create suitable graph sets even for encrypted C&C connections. One could also manually label connections.

Of course, there are also graphs for which we do not have a classification (that is, neither a C&C signature nor a benign signature has matched). These *unknown graphs* could be related to either malicious or benign traffic, and we do not consider them in the subsequent steps.

The **second step** uses the malicious and the benign graph sets as inputs and performs graph mining. More precisely, we use a graph mining technique, previously presented by Yan and Han [47,48], to identify subgraphs that frequently appear in the malicious graph set. These frequent subgraphs are likely to constitute the core activity linked to C&C connections. Some post-processing is then applied to compact the set of mined subgraphs. Finally, the set difference is computed between the mined, malicious subgraphs and the benign graph set. Only subgraphs that never appear by subgraph isomorphism in the benign graph set are selected. The assumption is that the selected subgraphs represent some host- and network-level activity that is only characteristic of particular C&C connections, but not benign traffic.

In [13], the authors used a similar approach to distinguish between malware and harmless programs. To this end, the authors used a leap mining technique presented by Yan et al. [46] that selects subgraphs which maximize the information gain between the malicious and benign graph sets, that is to say subgraphs that maximally cover (detect) the entire collection of malicious graphs while introducing a very low number of false positives. However, during the mining process, this technique tends to remove the graph parts that could be common to both benign and malicious graphs. In our present case, these parts are critical to obtain complete C&C templates. For example, in the case of a download and execute command, if the download part of the graph is observed in the benign set, leap mining would only mine the execute part. For these reasons, we performed the set difference with the benign graph set only as post-processing, once complete malicious subgraphs have already been mined, without risk of losing parts of them.

In addition, the algorithm proposed in [13] does not attempt to synthesize any semantic information from the mined behaviors; it does not produce a template that combines related behaviors and generalizes their common core. In other words [13], "this synthesis step does not add new behaviors to the set, it only combines the ones previously mined." In this paper, we go further and introduce two additional, novel steps to generalize the results obtained during the graph mining step. This is important because we want to generalize from specific instances of implementing a C&C connection and ab-

stract a core that characterizes the common and necessary operations for a particular type of command.

As a **third step**, we cluster the graphs previously mined. The goal of this step is to group together graphs that correspond to a similar type of command and control activity. That is, when we have observed different instances of one particular behavior, we combine the corresponding graphs into one cluster. As an example, consider different instances of a malware family where each sample downloads data from the network via HTTP, decodes it in some way, stores the data on disk, and finally executes that file. All instances of this behavior are examples for typical bot update mechanisms (*download and execute*), and we want to group all of them into one cluster. As a result of this step, we obtain different clusters, where each cluster contains a set of graphs that correspond to a particular C&C activity.

In the **fourth step**, we produce a single *C&C template* for each cluster. The goal of a template is to capture the common core of the graphs in a cluster; with the assumption that this common core represents the key activities for a particular behavior. The C&C templates are generated by iteratively computing the *weighted minimal common supergraph (WMCS)* [3] between the graphs in a cluster. The nodes and edges in the supergraph that are present in all individual graphs become part of the core. The remaining ones become optional.

At the end of this step, we have extracted templates that match the core of the program activities for different types of commands, taking into account optional operations that are frequently (but not always) present. This allows us to match variants of C&C traffic that might be different (to a certain degree) from the exact graphs that we used to generate the C&C templates.

3 System Details

In this section, we provide an overview of the actual implementation of JACKSTRAWS and explain the different analysis steps in greater details.

3.1 Analysis Environment

We use the dynamic malware analysis environment Anubis [20] as the basis for our implementation, and implemented several extensions according to our needs. Note that the general approach and the concepts outlined in this paper are independent of the actual analysis environment; we could have also used BitBlaze, Ether, or any other dynamic malware analysis environment.

As discussed in Section 2, behavior graphs are used to capture and represent the host-based activity that malware performs. To create such behavior graphs, we execute a malware sample and record the system calls that this sample invokes. In addition, we identify dependencies between different events of the execution by making use of dynamic taint analysis [38], a technique that allows us to asses whether a register or memory value depends on the output of a certain operation. Anubis already comes with tainting propagation support. By default, all output arguments of system calls from the native Windows API (e.g., `NtCreateFile`, `NtCreateProcess`, etc.) are marked with a unique taint label. Anubis then propagates the taint information while the monitored system processes tainted data. Anubis also monitors if previously tainted data is used as an input argument for another system call.

While Anubis propagates taint information for data in memory, it does not track taint information on the file system. In other words, if tainted data is written to a file and subsequently read back into memory, the original taint labels are not restored. This shortcoming turned out to be a significant drawback in our settings: For example, bots frequently download data from the C&C, decode it in memory, write this data to a file, and later execute it. Without taint tracking through the file system, we cannot identify the dependency between the data that is downloaded and the file that is later executed. Another example is the use of configuration data: Many malware samples retrieve configuration settings from their C&C servers, such as URLs that should be monitored for sensitive data or address lists for spam purposes. Such configuration data is often written to a dedicated file before it is loaded and used later. Restoring the original taint labels when files are read ensures that the subsequent bot activity is linked to the initial network connection and improves the completeness of the behavior graphs.

Finally, we improved the network logging abilities of Anubis by hooking directly into the Winsock API calls rather than considering only the abstract interface (`NtDeviceIOControlFile`) at the native system call level. This allows us to conveniently reconstruct the network flows, since send and receive operations are readily visible at the higher-level APIs.

3.2 Behavior Graph Generation

When the sample and all of its child processes have terminated, or after a fixed timeout (currently set to 4 minutes), JACKSTRAWS saves all monitored system calls, network-related data, and tainting information into a log file. Unlike previous work that used behavior graphs for distinguishing between malicious and legitimate programs, we use these graphs to determine the purpose of network connections (and to detect C&C traffic). Thus, we are not interested in the entire activity of the malware program. Instead, we only focus on actions related to network traffic. To this end, we first identify all send and receive operations that operate on a successfully-established network connection. In this work, we focus only on TCP traffic, and a connection is considered

successful when the three-way handshake has completed and at least one byte of user data was exchanged. All system calls that are related to a single network connection are added to the behavior graph for this connection. That is, for each network connection that a sample makes, we obtain one behavior graph which captures the host-based activities related to this connection.

For each send operation, we check whether the sent data is tainted. If so, we add the corresponding system call that produced this data to the behavior graph and connect both nodes with an edge. Likewise, for each receive operation, we taint the received data and check if it is later used as input to a system call. If so, we also add this system call to the graph and connect the nodes.

For each system call that is added to the graph in this fashion, we also check backward dependencies (that is, whether the system call has tainted input arguments). If this is the case, we continue to add the system call(s) that are responsible for this data. This process is repeated recursively as long as there are system calls left that have tainted input arguments that are unaccounted for. That is, for every node that is added to our behavior graph, we will also add all parent nodes that produce data that this node consumes. For example, if received data is written to a local file, we will add the corresponding `NtWriteFile` system call to the graph. This write system call will use as one of its arguments a file handle. This file handle is likely tainted, because it was produced by a previous invocation of `NtCreateFile`. Thus, we also add the node that corresponds to this create system call and connect the two nodes with an edge. On the other hand, forward dependencies are not recursively followed to avoid an explosion in the graph size.

Graph labeling. Nodes and edges that are inserted into the behavior graph are augmented with additional labels that capture more information about the nature of the system calls and the dependencies between nodes. For edges, the label stores either the names of the input or the output arguments of the system calls that are connected by a data dependency. For nodes, the label stores the system call name and some additional information that depends on the specific type of call. The additional information can store the type of the resource (files, registry keys, ...) that a system call operates on as well as flags such as mode or permission bits. Note that some information is only stored as comment; this information is ignored for the template generation and matching, but is saved for a human analyst who might want to examine a template.

One important additional piece of information stored for system calls that manipulate files and registry keys is the name of these files and keys. However, for these resource names, it is not desirable to use the actual string. The reason is that labels are taken into account during the matching process, and two nodes are considered the same only when their labels match. Thus, some type of abstraction is necessary for labels that represent resource names, otherwise, graphs become too specific. We generalize file names based on the location of the file (using the path name) and its type (typically, based on the file's extension). Registry key names are generalized by normalizing the key root (using abbreviations) and replacing random names by a generic format (typically, numerical values). More details about the labeling process and these abstractions can be found in Appendix A.

Simplifying behavior graphs. One problem we faced during the behavior graph generation was that certain graphs grew very large (in terms of number of nodes), but the extra nodes only carried duplicate information. For example, consider a bot that downloads an executable file. When this file is large, the data will not be read from the network connection by a single `recv` call. Instead, the receive system call might be invoked many times; in fact, we have observed samples that read network data one byte at a time. Since every system call results in a node being added to the behavior graph, this can increase the number of nodes significantly.

To reduce the number of (essentially duplicate) nodes in the graph, we introduce a post-processing step that collapses certain nodes. The purpose of this step is to combine multiple nodes, sharing the same label and dependencies. More precisely, for each pair of nodes with an identical label in the behavior graph, we check whether (1) the two nodes share the same set of parent nodes, or (2) the sets of parents and children of one node are respective subsets of the other, or (3) one node is the only parent of the other. If this is the case, we collapse these nodes into a single node and add a special tag *Is-Multiple* to the label. Additional incoming and outgoing edges of the aggregated nodes are merged into the new node. The process is repeated until no more collapsing is possible. As an example, consider the case where a write file operation stores data that was previously read from the network by multiple receive calls. In this case, the write system call node will have many identical parent nodes (the receive operations), which all contribute to the buffer that is written. In the post-processing step, these nodes are all merged into a single system call. A beneficial side-effect of node collapsing is that this does not only reduce the number of nodes, but also provides some level of abstraction from the concrete implementation of the malware code and the number of times identical functions are called (as part of a loop, for example).

Summary. The output of the two previous steps is one behavior graph for each network connection that a malware sample makes. Behavior graphs can be used in two ways: First, we can match behavior graphs, produced by running unknown malware samples, against a set

of C&C templates that characterize malicious activity. When a template matches, the corresponding network connection can be labeled as command and control. This matching procedure is explained in Section 3.6.

The second use of behavior graphs is for C&C template generation. For this process, we assume that we know some connections that are malicious and some that are benign. We can then extract the subgraphs from the behavior graphs that are related to known malicious C&C connections and subgraphs that represent benign activity. These two sets of malicious and benign graphs form the input for the template generation process that is described in the following three sections.

3.3 Graph Mining

The first step when generating C&C templates is graph mining. More precisely, the goal is to mine frequent subgraphs that are *only* present in the malicious set. An overview of the process can be seen in Figure 2.

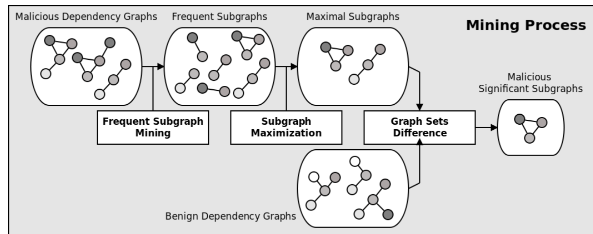


Figure 2: Mining process.

Frequent subgraphs are those that appear in more than a fraction k of all malicious graphs. When k is too high, we might miss many interesting behaviors (subgraphs) that are not frequent enough to exceed this threshold. When k is too low, more behaviors are covered, but unfortunately, the mining process will produce such a massive amount of graphs that it never terminates. We discuss the concrete choice of k in Section 4.

Frequent subgraph mining. There exist a number of tools that can be readily used for mining frequent subgraphs. For this paper, we decided to use *gSpan* [47,48] because it is stable, and supports labeled graphs, both at the node and edge level. *gSpan* relies on a lexicographic ordering of graphs and uses a depth-first search strategy to efficiently mine frequent, connected subgraphs.

A limitation of *gSpan* is that it only supports undirected edges, whereas behavior graphs are, by nature, directed since the edges represent data flows. To work around this limitation and produce directed subgraphs, we encode the direction of edges into their labels, and then restore the direction information at the end of the mining process. Moreover, *gSpan* accepts only numeric values as labels for nodes and edges. Thus, we cannot directly use the string labels (names or flags) that are associated with nodes and edges in the behavior graphs. To solve this, we simply concatenate all string labels of a node or edge and hash the result. Then, this hash value is mapped into a unique integer.

Subgraph maximization. The output produced by *gSpan* contains many graphs that are subgraphs of others. The reason is that *gSpan* works by growing subgraphs. That is, it first looks for individual nodes that are frequent. Then, *gSpan* adds one additional node and re-runs the frequency checks. This *add-and-check* process is repeated until no more frequent graphs can be found. However, during this process, *gSpan* outputs all subgraphs that are frequent. Thus, the result of the mining step contains all intermediate subgraphs whose frequency is above the selected threshold.

Unfortunately, these redundant, intermediate subgraphs negatively affect the subsequent template generation steps because they distort the frequencies of nodes and edges. To solve this problem, we introduce a maximization step. The purpose of this step is to remove a subgraph G_{sub} if there exists a supergraph G_{super} in the same result set that contains G_{sub}. Looking at Figure 2, the result of the maximization step is that all 2-node graphs are removed because they are subgraphs of the 3-node graphs. However, removing subgraphs is not always desirable: even when both a subgraph G_{sub} and a supergraph G_{super} exceed the frequency threshold k, the subgraph G_{sub} might be much more frequent than G_{super}. In this case, both graphs should be kept. To this end, we only remove a subgraph G_{sub} when its frequency is less than twice the frequency of G_{super}.

Graph sets difference. So far, we have mined graphs that frequently appear in the malicious set. However, we also require that these graphs do *not* appear in the benign set. Otherwise, they would not be suitable to distinguish C&C connections from other traffic.

To remove graphs that are present in the benign set, we compute the set difference between the frequent malicious subgraphs and benign graphs. More precisely, we use a sub-isomorphism test to determine, for each malicious graph, whether it appears in some benign graphs. If this is the case, it is removed from the mining results. Looking at the example in Figure 2, the set difference removes one graph that also appears in the benign set. As an interesting technical detail, our approach of using set difference to obtain interesting, malicious subgraphs is different from the technique presented in [13]. In [13], the authors use leap mining, which operates simultaneously on the malicious and benign sets to find graphs with a high frequency within the malicious set and a low frequency within the benign set [46].

By construction, leap mining removes *all* parts from the output that are shared between benign and malicious graphs. For example, consider a behavior graph that cap-

tures a command that downloads data, stores it to a file, and later executes this file. If the download part of this graph is also present in the benign set, which is likely to be the case (since downloading data is not malicious *per se*), this part will be removed. Thus, the malicious graph will only contain the part where the downloaded file is executed. That is, in this example, leap mining would produce an incomplete graph that covers only part of the relevant, malicious activity. In our case, we first generate the entire graph that captures both the download and the execute. Then, the set difference algorithm checks whether this entire graph occurs also in the benign set. Since no benign graph is presumably a supergraph of the malicious behavior, the entire graph is retained.

3.4 Graph Clustering

Using as input the frequent, malicious subgraphs produced by the previous mining step, the purpose of this step is to find clusters of similar graphs (see Figure 3). The graph mining step produces different graphs that represent different types of behaviors. We now need to cluster these graphs to find groups, where each group shares a common core of activities (system calls) typical of a particular behavior. Graph clustering is used for this purpose; generated clusters are later used to create generalized templates covering the graphs they contain.

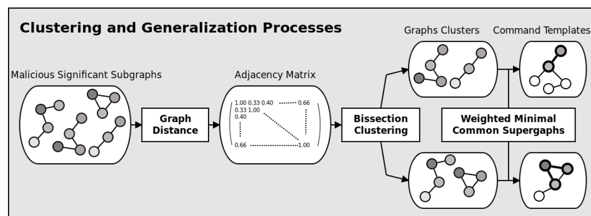


Figure 3: Clustering and generalization processes.

A crucial component for every clustering algorithm is the proper choice of a similarity measure that computes the distances between graphs. In our system, the similarity measure between two graphs is based on their *non-induced, maximum common subgraph (mcs)*. The *mcs* of two graphs is the maximal subgraph that is isomorphic to both. The intuition behind the measure is the following: We expect two graphs that represent the same malware behavior to share a common core of nodes that capture this behavior. The *mcs* captures this core. Thus, the *mcs* will be large for similar graphs. From now on, all references to the *mcs* will refer to the non-induced construction. The similarity measure is defined as:

$$d(G_1, G_2) = \frac{2 \times |edges(mcs(G_1, G_2)|}{|edges(G_1)| + |edges(G_2)|} \qquad (1)$$

To compute the *mcs* between two graphs, we use the McGregor backtracking algorithm [6]. According to benchmarking results [6], this algorithm performs well on randomly-connected graphs with small density. In our case, behavior graphs have no cycles and only a limited number of dependencies; this is close to randomly-connected graphs rather than regular or irregular meshes.

As shown in Figure 3, we use the *mcs* similarity measure to compute the one-to-one distance matrix between all mined graphs. We then use a tool, called *Cluto* [24], to find clusters of similar graphs. *Cluto* implements a variety of different clustering algorithms; we selected *clustering by repeated bisection*. This algorithm works as follows: All graphs are originally put into a single cluster. This cluster is then iteratively split until the similarity in each sub-cluster is larger than a given similarity threshold [24, 50]. At each step, the cluster to be split is chosen so that the similarity function between the elements of that clusters is maximized. The advantage of this technique is that we do not need to define a fixed number of clusters *a priori*. Moreover, one also does not need to select initial graphs as center to build the clusters around (as with k-means clustering). The output of this step is a set of clusters that contain similar graphs.

3.5 Graph Generalization and Templating

Based on the clusters of similar graphs, the final step in our template generation process is graph generalization (the rightmost step in Figure 3). The goal of the generalization process is to construct a template graph that abstracts from the individual graphs within a cluster. Intuitively, we would expect that a template contains a *core* of nodes, which are common to all graphs in a cluster. In addition, to capture small differences between these graphs, there will be *optional nodes* attached to the core.

The generalization algorithm computes the *weighted minimal common supergraph (WMCS)* of all the graphs within a given cluster [3]. The *WMCS* is the minimal graph such that all the graphs of the cluster are contained within it. To distinguish between core and optional nodes, we use weights. These weights capture how frequent a node or an edge in the *WMCS* is present in one of the individual graphs. For core edges and core nodes, we expect that they are present in all graphs of a cluster (that is, their weight is n in the *WMCS*, assuming that there are n graphs in the cluster). All other nodes with a weight smaller than n become optional nodes.

The approach to compute a template is presented in Algorithm 1. The *WMCS* is first initialized with the first graph G_1 of the cluster, and the weights of all its nodes and edges are set to 1. The integration of an additional graph G_i is performed as follows: We first determine the maximal common subgraph *mcs* between G_i and the current *WMCS*. The nodes and edges in the *WMCS* that are part of the *mcs* have their weight increased by 1. The

Algorithm 1 *Weighted minimum common supergraph*

Require: A graph set $G_1, ..., G_n$
1: $WMCS \leftarrow G_1$
2: $\forall n \in nodes(T)$ **and** $e \in edges(T)$ **do** $w_n := 1$ **and** $w_e := 1$
3: **for** $i = 2$ **to** n **do**
4: $s := state_exploration(\oslash)$
5: $mcs \leftarrow maximum_common_subgraph(G_i, WMCS, s)$
6: $\forall n \in nodes(mcs)$ **do** $w_n \mathrel{+}= 1$
7: $\forall e \in edges(mcs)$ **do** $w_e \mathrel{+}= 1$
8: $\forall n \in nodes(G_i)$ **and** $n \notin nodes(mcs)$, **do** $WMCS.add_node(n)$ **and** $w_n := 1$
9: $\forall e \in edges(G_i)$ **and** $e \notin edges(mcs)$, **do** $WMCS.add_edge(e)$ **and** $w_e := 1$
10: **end for**
11: **return** $WMCS$

nodes and edges in G_i that are not part of the *mcs* are added to the *WMCS*, and their weight is set to 1.

To increase the generality of a template, the labels of optional nodes are further relaxed. More precisely, our system preserves the label that stores the name of the system call. However, all additional information is replaced by a wild card that matches every possible, concrete parameter later. Finally, we remove all templates with a core of three or fewer nodes. The reason is that these templates are likely too small to accurately capture the entire malicious activity and might lead to false positives. In the example in Figure 3, core nodes and edges are shown as dark elements, while the optional elements are white. We generate one C&C template per cluster.

3.6 Template Matching

The previous steps leveraged machine learning techniques and sets of known malicious and benign graphs to produce a number of C&C templates. These templates are graphs that represent host-based activity that is related to command and control traffic. To find the C&C connections for a new malware sample, this sample is first executed in the sandbox, and our system extracts its behavior graphs (as discussed in Sections 3.1 and 3.2). Then, we match all C&C templates against the behavior graphs. Whenever a match is found, the corresponding connection is detected as command and control traffic, and we can extract its endpoints (IPs and domains).

The matching technique is described in Algorithm 2. In a first step, we attempt to determine whether the core of a template T is present in the behavior graph G. To this end, we simply use a subgraph isomorphism test. When the test fails, we know that the core nodes of T are not part of the graph, and we can advance to trying the next template. If the core is found, we obtain the mapping from the core nodes to the corresponding nodes in G. We then test the optional nodes. To this end, we compute the *mcs* between T and G. For this, the fixed mapping provided by the previous isomorphism test is used to initialize the space exploration when building the

Algorithm 2 *Template matching*

Require: A behavior graph G, A template T
1: $map \leftarrow subgraph_isomorphism(core(T), G)$
2: **if** $map = \oslash$ **then**
3: **return** $false$
4: **end if**
5: $s := state_exploration(map)$
6: $mcs \leftarrow maximum_common_subgraph(G, T, s)$
7: **return** $true, mcs$

mcs, significantly speeding up the process. Based on the result of the *mcs* computation, we can directly see how many optional nodes have matched, that is to say, are covered by the *mcs*. Taking into account the fraction (or the absolute number) of optional nodes that are found in G, we can declare a template match.

4 Evaluation

Experiments were performed to evaluate JACKSTRAWS both from a quantitative and qualitative perspective. This section describes the evaluation details and results.

4.1 Evaluation Datasets

For the evaluation, our system analyzed a total of 37,572 malware samples. The samples were provided to us by a network security company, who obtained the binaries from recent submission to a public malware analysis sandbox. Moreover, we were only given samples that showed some kind of network activity when run in the sandbox. We were also provided with a set of 385 signatures specifically for known C&C traffic, as well as 162 signatures that characterize known, benign traffic. As mentioned previously, the company uses signatures for benign traffic to be able to quickly discard harmless connections that bots frequently make.

To make sure that our sample set covers a wide variety of different malware families, we labeled the entire set with six different anti-virus engines: Kaspersky, F-Secure, BitDefender, McAfee, NOD32, and F-Prot. Using several sources for labeling allows us reduce the possible limitations of a single engine. For every malware sample, each engine returns a label (unless the samples is considered benign) from which we extract the malware family substring. For instance, if one anti-virus engine classifies a sample as *Win32.Koobface.AZ*, then *Koobface* is extracted as the family name. The family that is returned by a majority of the engines is used to label a sample. In case the engines do not agree (and there is no majority for a label), we go through the output of the AV tools in the order that they were mentioned previously and pick the first, non-benign result.

Overall, we identified 745 different malware families for the entire set. The most prevalent families were *Generic* (3756), *EgroupDial* (2009), *Hotbar* (1913), *Palevo* (1556), and *Virut* (1539). 4,096 samples re-

mained without label. Note that *Generic* is not a precise label; many different kinds of malware can be classified as such by AV engines. In summary, the results indicate that our sample set has no significant bias towards a certain malware family. As expected, it covers a rich and diverse set of malware, currently active in the wild.

In a first step, we executed all samples in JACKSTRAWS. Each sample was executed for four minutes, which allows a sample to initialize and perform its normal operations. This timeout is typically enough to establish several network connections and send/receive data via them. The execution of the 37,572 samples produced 150,030 network connections, each associated with a behavior graph. From these graphs, we removed 19,395 connections in which the server responded with an error (e.g., an HTTP request with a 404 "Not Found" response). Thus, we used a total of 130,635 graphs produced by a total of 33,572 samples for the evaluation.

In the next step, we applied our signatures to the network connections. This resulted in 16,535 connections that were labeled as malicious (known C&C traffic, 12.7%) and 16,082 connections that were identified as benign (12.3%). The malicious connections were produced by 9,108 samples, while the benign connections correspond to 7,031 samples. The remaining 98,018 connections (75.0%) are unknown. The large fraction of unknown connections is an indicator that it is very difficult to develop a comprehensive set of signatures that cover the majority of bot-related C&C traffic. In particular, there was at least one unclassified connection for 31,671 samples. Note that the numbers of samples that produced malicious, benign, and unknown traffic add up to more than the total number of samples. This is because some samples produced both malicious and benign connections. This underlines that it is difficult to pick the important C&C connections among bot traffic.

Of course, not all of the 385 malicious signatures produced matches. In fact, we observed only hits from 78 C&C signatures, and they were not evenly distributed. A closer examination revealed that the signature that matched the most number of network connections is related to *Palevo* (4,583 matches), followed by *Ramnit* (3,896 matches) and *Koobface* (2,690 matches).

4.2 Template Generation

Initially, we put all 16,535 behavior graphs that correspond to known C&C connections into the *malicious graphs set*, while the 16,082 graphs corresponding to benign connections were added to the *benign graphs set*. To improve the quality of these sets, we removed graphs that contained too few nodes, as well as graphs that contained only nodes that correspond to network-related system calls (and a few other house-keeping functions that are not security-relevant). Moreover, to maintain a balanced training set, we kept at most three graphs (connections) for each distinct malware sample. This pre-processing step reduced the number of graphs in the malicious set to 10,801, and to 12,367 in the benign set.

Both sets were then further split into a training set and a test set. To this end, we randomly picked a number of graphs for the training set, while the remaining ones were set aside as a test set. More precisely, for the malicious graphs, we kept 6,539 graphs (60.5%) for training and put 4,262 graphs (39.5%) into the test set. For the benign graphs, we kept 8,267 graphs (66.8%) for training and put 4,100 graphs (33.2%) into the test set. We used these malicious and benign training sets as input for our template generation algorithm. This resulted in 417 C&C templates that JACKSTRAWS produced. The average number of nodes in a template was 11, where 6 nodes were part of the core and 5 were optional.

For the mining process, we used a threshold $k = 0.1$. That is, the mining tool will pick subgraphs from the training sets only when they appear in more than 10% of all behavior graphs. The reason why we could operate with a relatively large threshold of $k = 0.1$ is that we divided the behavior graphs into different bins, and mined on each bin individually. To divide graphs into bins, we observe that certain malware activity requires the execution of a particular set of system calls. For example, to start a new process, the malware needs to call `NtCreateProcess`, or to write to a file, it needs to call `NtWriteFile`. Thus, we selected five security-relevant system activities (registry access; file system access; process creation; queries to system information; and accesses to web-related resources, such as HTML or JS files) and assigned each to a different bin. Then, we put into each bin all behavior graphs that contain a node with the corresponding activity (system calls). Graphs that did not fall into any of these bins were gathered in a miscellaneous bin. It is important to observe that this step merely allows us to mine with a higher threshold, and thus to accelerate the graph mining process considerably. We would have obtained the same set of templates (and possibly more) when mining on the entire training set with a lower mining threshold.

For the clustering process, we iterated the bisection operation until the average similarity within the clusters was over 60% and the minimal similarity was over 40%. Higher thresholds were discarded because they increased the number of clusters, making them too specific.

Producing templates for the 14,806 graphs in the training set took about 21 hours on an Intel Xeon 4 CPUs 2.67GHz server, equipped with 16GB of RAM. This time was divided into 16 hours for graph mining, 4.5 hours for clustering, and 30 minutes for graph generalization. This underlines that, despite the potentially

costly (NP-hard) graph algorithms, JACKSTRAWS is able to efficiently produce results on a large, real-world input dataset. The mining process was the most time-consuming operation, but the number of mined subgraphs was, in the end, five times smaller than the number of graphs in input. Consequently, the clustering process, which is polynomial in function of the number of graphs in input, ran on a reduced set. For the template generation process, the resulting clusters only contained 10 to 20 graphs on average, explaining the faster computations.

4.3 Detection Accuracy

In the next step, we wanted to assess whether the generated templates can accurately detect activity related to command and control traffic without matching benign connections. To this end, we ran two experiments. First, we evaluated the templates on the graphs in the test set (which correspond to known C&C connections). Then, we applied the templates to graphs associated with unknown connections. This allows us to determine whether the extracted C&C templates are generic enough to allow detection of previously-unknown C&C traffic (for which no signature exists).

Experiment 1: Known C&C connections. For the first experiment, we made use of the test set that was previously set aside. More precisely, we applied our 417 templates to the behavior graphs in the test set. This test set contained 4,262 connections that matched C&C signatures and 8,267 benign connections.

Our results show that JACKSTRAWS is able to successfully detect 3,476 of the 4,262 malicious connections (81.6%) as command and control traffic. Interestingly, the test set also contained malware families that were absent from the malicious training set. 51.7% of the malicious connections coming from these families were successfully detected, accounting for 0.4% of all detections. While the detection accuracy is high, we explored false negatives (i.e., missed detections) in more detail. Overall, we found three reasons why certain connections were not correctly identified:

First, in about half of the cases, detection failed because the bot did not complete its malicious action after it received data from the C&C server. Incomplete behavior graphs can be due to a timeout of the dynamic analysis environment, or an invalid configuration of the host to execute the received command properly.

Second, the test set contained a significant number of *Adware* samples. The behavior graphs extracted from these samples are very similar to benign graphs; after all, *Adware* is in a grey area different from malicious bots. Thus, all graphs potentially covering these samples are removed at the end of the mining process, when compared to the benign training sets.

The third reason for missed detections are malicious connections that are only seen a few times (possibly only in the test set). According to the AV labels, our data set covers 745 families (and an additional 4,096 samples that could not be labeled). Thus, certain families are rare in the data set. When a specific graph is only present a few times (or not at all) in the training set, it is possible that all of its subgraphs are below the mining threshold. In this case, we do not have a template that covers this activity.

JACKSTRAWS also reported 7 benign graphs as malicious out of 4,100 connections in the benign test set: a false positive rate of 0.2%. Upon closer examination, these false positives correspond to large graphs where some Internet caching activity is observed. These graphs accidentally triggered four weaker templates with few core and many optional nodes.

Overall, our results demonstrate that the host-based activity learned from a set of known C&C connections is successful in detecting other C&C connections that were produced by a same set of malware families, but also in detecting five related families that were only present in the test set. In a sense, this shows that C&C templates have a similar detection capability as manually-generated, network-based signatures.

We also wanted to understand the impact of template generalization compared to previous work that used directly the mined subgraphs [13]. For this, we used the graphs mined from the malicious training set as signatures, without any generalization (this is the approach followed in previous work). Using a sub-isomorphism test for detection over the 4,262 malicious graphs in the test set, we found that the detection rate was 66%, 15.6% lower. This underlines that the novel template generation process provides significant benefits.

Experiment 2: Unknown connections. For the next experiment, we decided to apply our templates to the graphs that correspond to unknown network traffic. This should demonstrate the ability of JACKSTRAWS to detect novel C&C connections within protocols not covered by any network-level signature.

When applying our templates to the 98,018 unknown connections, we found 9,464 matches (9.7%). We manually examined these connections in more detail to determine whether the detection results are meaningful. The analysis showed that our approach is promising; the vast majority of connections that we analyzed had clear indications of C&C activity. With the help of the anti-virus labels, we could identify 193 malware families which were *not* covered by the network signatures. The most prevalent new families were *Hotbar* (1984), *Pakes* (871), *Kazy* (107), and *LdPinch* (67). Furthermore, we detected several new variants of known bots that we did not detect previously because their network fingerprint

had changed and, thus, none of our signatures matched. Nevertheless, JACKSTRAWS was able to identify these connections due to matched templates. In addition, the manual analysis showed a low number of false positives. In fact, we only found 27 false positives out of the 9,464 matches, all of them being HTTP connections.

When comparing the number of our matches with the total number of unknown connections, the results may appear low at first glance. However, not all connections in the unknown set are malicious. In fact, 10,524 connections (10.7%) do not result in any relevant host-activity at all (the graphs only contain network-relayed system calls such as `send` or `connect`). For another 13,676 graphs (14.0%), the remote server did not send any data. For more than 7,360 HTTP connections (7.5%), the server responded with status code 302, meaning that the requested content had moved. In this case, we probably cannot see any interesting behavior to match. In a few hundred cases, we also observed that the timeout of JACKSTRAWS interrupted the analysis too early (e.g., the connection downloaded a large file). In these cases, we usually miss some of the interesting behavior. Thus, almost 30 thousand unknown connections can be immediately discarded as non-C&C traffic.

Furthermore, the detection results of 9,464 new C&C connections for JACKSTRAWS need to be compared with the total number of 16,535 connections that the entire signature set was able to detect. Our generalized templates were able to detect almost 60% more connections than hundreds of hand-crafted signatures. Note that our C&C templates do not inspect network traffic at all. Thus, they can, by construction, detect C&C connections regardless of whether the malware uses encryption or not, something not possible with network signatures.

4.4 Template Quality

The previous section has shown that our C&C templates are successful in identifying host-based activity related to both known and novel network connections. We also manually examined several templates in more detail to determine whether they capture activity that a human analyst would consider malicious.

JACKSTRAWS was able to extract different kinds of templates. A few template examples are shown in Appendix B. More precisely, out of the 417 templates, more than a hundred templates represent different forms of information leakage. The leaked information is originally collected from dedicated registry keys or from specific system calls (e.g., computer name, Windows version and identifier, Internet Explorer version, current system time, volume ID of the hard disk, or processor information). About fifty templates represent executable file downloads or updates of existing files. Additional templates include process execution: downloaded data that is injected into a process and then executed. Five templates also represent complete download and execute commands. The remaining templates cover various other malicious activities, including registry modifications ensuring that the sample is started on certain events (e.g., replacing the default executable file handler for Windows Explorer) and for hiding malware activity (e.g., clearing the MUICache).

We also found 20 "weak" templates (out of 417). These templates contain a small number of nodes and do not seem related to any obvious malicious activity. However, these templates did not trigger any false positive in the benign test set. This indicates that they still exhibit enough discriminative power with regards to our malicious and benign graph sets.

5 Related Work

Given the importance and prevalence of malware, it is not surprising that there exists a large body of work on techniques to detect and analyze this class of software. The different techniques can be broadly divided into host-based and network-based approaches, and we briefly describe the related work in the following.

Host-based detection. Host-based detection techniques include systems such as traditional anti-virus tools that examine programs for the presence of known malware. Other techniques work by monitoring the execution of a process for behaviors (e.g., patterns of system calls [12, 28, 32]) that indicate malicious activity. Host-based approaches have the advantage that they can collect a wealth of detailed information about a program and its execution. Unfortunately, collecting a lot of information comes with a price; it incurs a significant performance penalty. Thus, detailed but costly monitoring is typically reserved for malware analysis, while detection systems, which are deployed on end-user machines, resort to fast but imprecise techniques [43]. As a result, current anti-virus products show poor detection rates [4].

A suitable technique to model the host-based activity of a program is a behavior graph. This approach has been successfully used in the past [5, 13, 26] and we also apply this technique. Recently, Fredrikson et al. introduced an approach to use graph mining on behavior graphs in order to distinguish between malicious and benign programs [13]. Graph mining itself is a well-known technique [46–48] that we use as a building block of JACKSTRAWS. Compared to their work, we have another high-level goal: we want to learn which network connections are related to C&C traffic in an automated way. Thus we do not only focus on host-level activities, but also take the network-level view into account and correlate both. Furthermore, we also cluster the graphs and perform a generalization step to extract templates that describe the characteristics of C&C connections.

From a technical point of view, we perform a more fine-grained analysis by applying taint analysis instead of the coarse-grained analysis performed by [13].

BOTSWAT [41] analyzes how bots process network data by analyzing system calls and performing taint analysis. The system matches the observed behavior against a set of 18 manually generated behavior patterns. In contrast, we use mining and machine learning techniques to automatically generate C&C templates. From a technical point of view, BOTSWAT uses library-call-level taint analysis and, thus, might miss certain dependencies. In contrast, the data flow analysis support of JACKSTRAWS enables a more fine grained analysis of information flow dependency among system calls.

Network-based detection. To complement host-based systems and to provide an additional layer for defense-in-depth, researchers proposed network-based detection techniques [15–18, 45, 49]. Network-based approaches have the advantage that they can cover a large number of hosts without requiring these hosts to install any software. This makes deployment easier and incurs no performance penalty for end users. On the downside, network-based techniques have a more limited view (they can only examine network traffic and encryption makes detection challenging), and they do not work for malicious code that does not produce any network traffic (which is rarely the case for modern malware).

Initially, network-based detectors focused on the artifacts produced by worms that spread autonomously through the Internet. Researchers proposed techniques to automatically generate payload-based signatures that match the exploits that worms use to compromise remote hosts [25,27,29,31,39]. With the advent of botnets, malware authors changed their *modus operandi*. In fact, bots rarely propagate by scanning for and exploiting vulnerable machines; instead, they are distributed through drive-by download exploits [36], spam emails [22], or file sharing networks [23]. However, bots do need to communicate with a command and control infrastructure. The reason is that bots need to receive commands and updates from their controller, and also upload stolen data and status information. As a result, researchers shifted their efforts to developing ways that can detect and disrupt malicious traffic between bots and their C&C infrastructure. In particular, researchers proposed approaches to identify (and subsequently block) the IP addresses and domains that host C&C infrastructures [42], techniques to generate payload signatures that match C&C connections [15, 17, 45], and anomaly-based systems to correlate network flows that exhibit a behavior characteristic of C&C traffic [16, 18, 49]. In a paper related to ours, Perdisci et al. studied how network traces of malware can be clustered to identify families of bots that perform similar C&C communication [34]. The clustering results can be used to generate signatures, but their approach does not take into account that bots generate benign traffic or can even deliberately inject noise [1, 10, 11, 33]. Our work is orthogonal to this approach since we can precisely identify connections related to C&C traffic.

6 Limitations

We aim at analyzing malicious software, which is a hard task in itself. An attacker can use different techniques to interfere with the analysis environment which is of concern for us. Our approach relies on actually observing the network communication of the sample to build the corresponding behavior graph. Thus, we need to consider attacks against the dynamic analysis environment, and, specifically, the taint analysis, since this component allows us to analyze the interdependence of network and host activities. Several techniques have been introduced in the past to enhance the analysis capabilities, for example, multi-path execution [30] or the analysis of VM-aware samples [2]. These and similar methods can be integrated in JACKSTRAWS so that the dynamic analysis process produces more extensive analysis reports. Note, however, that the evaluation results demonstrate that we can successfully, and in a large scale, analyze complex, real-world malware samples. This indicates that the prototype version of JACKSTRAWS already provides a robust framework for performing our analysis

Of course, an attacker might develop techniques to thwart our analysis, for example, by interleaving unnecessary system calls with the calls that represent the actual, malicious activity. The resulting, additional nodes might hinder the mining process and prevent the extraction of a graph core. An attacker might also try to introduce duplicate nodes to launch complexity attacks, since most of the graph algorithms used in JACKSTRAWS are known to be NP-complete [6]. However, interleaved calls have to share some data dependencies with relevant system calls, otherwise, they would be stripped from the behavior graph. Moreover, they must be specifically crafted to escape the collapsing mechanism. Another approach to disturb the analysis is to mutate the sequence of system calls that implement a behavior, as discussed in [21]. A possible solution to this kind of attacks is to normalize the behavior graphs in input using rewriting techniques. That is, semantically equivalent graph patterns are rewritten into a canonical form before mining.

7 Conclusion

In this paper, we focused on the problem of identifying actual C&C traffic when analyzing binary samples. During a dynamic analysis run, bots do not only communicate with their C&C infrastructure, but they often open also a large number of benign network connections. We introduced JACKSTRAWS, a tool that can identify C&C

traffic in a sound way. This is achieved by correlating network traffic with the associated host behavior.

With the help of experiments, we demonstrated the different templates we extracted and showed that we can even infer information about unknown bot families which we did not recognize before. On the one hand, we showed that our approach can be applied to proprietary protocols, which demonstrates that it is protocol agnostic. On the other hand, we also applied JACKSTRAWS to HTTP traffic, which is challenging since we need to reason about small differences between legitimate and malicious usage of the Windows API. The results show that we can still extract precise templates in this case.

8 Acknowledgments

This work was supported by the ONR under grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, Secure Business Austria, and the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (grant 315-43-02/2-005-WFBO-009). We also thank the anonymous reviewers for their comments that helped to improve the paper, Xifeng Yan for his precious help on graph mining, and Luca Foschini for his help on graph algorithms.

References

[1] S. Adair. Pushdo DDoS'ing or Blending In? `http://www.shadowserver.org/wiki/pmwiki.php/Calendar/20100129`, January 2010.

[2] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Symp. Network & Distributed System Security (NDSS)*, 2010.

[3] H. Bunke, P. Foggia, C. Guidobaldi, and M. Vento. Graph Clustering Using the Weighted Minimum Common Supergraph. In *Graph Based Representations in Pattern Recognition*, 2003.

[4] M. Christodorescu and S. Jha. Testing Malware Detectors. In *ACM Int. Symp. on Software Testing & Analysis (ISSTA)*, 2004.

[5] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Meeting of the European Software Engineering Conf. & the SIGSOFT Symp. Foundations of Software Engineering*, 2007.

[6] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms & Applications*, 11(1), 2007.

[7] E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *USENIX Workshop Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*, 2005.

[8] D. Dagon, G. Gu, C. Lee, and W. Lee. A Taxonomy of Botnet Structures. In *Annual Computer Security Applications Conf. (ACSAC)*, 2007.

[9] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: Malware Analysis Via Hardware Virtualization Extensions. In *ACM Conf. Computer & Communications Security (CCS)*, 2008.

[10] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *ACM Conf. Computer & Communications Security (CCS)*, 2006.

[11] P. Fogla, M. I. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic Blending Attacks. In *Usenix Security Symp.*, 2006.

[12] S. Forrest, S. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symp. Security & Privacy*, 1996.

[13] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symp. Security & Privacy*, 2010.

[14] F. C. Freiling, T. Holz, and G. Wicherski. Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In *European Symp. Research in Computer Security (ESORICS)*, 2005.

[15] J. Goebel and T. Holz. Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation. In *USENIX Workshop Hot Topics in Understanding Botnets (HotBots)*, 2007.

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In *USENIX Security Symp.*, 2008.

[17] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. In *USENIX Security Symp.*, 2006.

[18] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *Symp. Network & Distributed System Security (NDSS)*, 2008.

[19] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. C. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *Usenix Workshop Large-Scale Exploits & Emergent Threats (LEET)*, 2008.

[20] International Secure Systems Lab. Anubis: Analyzing Unknown Binaries. `http://anubis.iseclab.org`, 2011.

[21] G. Jacob, E. Filiol, and H. Debar. Functional polymorphic engines: formalisation, implementation and use cases. *Journal in Computer Virology*, 5(3):247–261, 2009.

[22] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2009.

[23] A. Kalafut, A. Acharya, and M. Gupta. A Study of Malware in Peer-to-Peer Networks. In *ACM SIGCOMM Conf. Internet Measurement*, 2006.

[24] G. Karypis. CLUTO - A Clustering Toolkit. Technical Report 02-017, University of Minnesota, 2003.

[25] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symp.*, 2004.

[26] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *USENIX Security Symp.*, 2009.

[27] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1), 2004.

[28] W. Lee, S. J. Stolfo, and K. W. Mok. A Data Mining Framework for Building Intrusion Detection Models. In *IEEE Symp. Security & Privacy*, 1999.

[29] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast Signature Generation for Zero-day PolymorphicWorms with Provable Attack Resilience. In *IEEE Symp. Security & Privacy*, 2006.

[30] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *IEEE Symp. Security & Privacy*, 2007.

[31] J. Newsom, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symp. Security & Privacy*, 2005.

[32] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Analysis of Computer Intrusions Using Sequences of Function Calls. *IEEE Trans. Dependable Secur. Comput.*, 4(2), 2007.

[33] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symp. Security & Privacy*, 2006.

[34] R. Perdisci, W. Lee, and N. Feamster. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *USENIX Symp. Networked Systems Design & Implementation (NSDI)*, 2010.

[35] P. Porras, H. Saïdi, and V. Yegneswaran. A Foray into Confickers Logic and Rendezvous Points. In *Usenix Workshop Large-Scale Exploits & Emergent Threats (LEET)*, 2009.

[36] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *USENIX Security Symp.*, 2008.

[37] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *Internet Measurement Conference (IMC)*, 2006.

[38] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symp. Security & Privacy*, 2010.

[39] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *USENIX Symp. Operating Systems Design & Implementation (OSDI)*, 2004.

[40] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Int. Conf. Information Systems Security (ICISS)*, 2008.

[41] E. Stinson and J. C. Mitchell. Characterizing Bots' Remote Control Behavior. In *Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2007.

[42] B. Stone-Gross, A. Moser, C. Kruegel, and E. Kirda. FIRE: FInding Rogue nEtworks. In *Annual Computer Security Applications Conf. (ACSAC)*, 2009.

[43] P. Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[44] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security & Privacy*, 5(2), 2007.

[45] P. Wurzinger, L. Bilge, T. Holz, J. Göbel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *European Symp. Research in Computer Security (ESORICS)*, 2009.

[46] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *ACM SIGMOD Int. Conf. Management of Data*, 2008.

[47] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *Int. Conf. Data Mining (ICDM)*, 2002.

[48] X. Yan and J. Han. CloseGraph: mining closed frequent graph patterns. In *ACM SIGKDD Int. Conf. Knowledge Discovery & Data Mining (KDD)*, 2003.

[49] T.-F. Yen and M. K. Reiter. Traffic Aggregation for Malware Detection. In *Conf. Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2008.

[50] Y. Zhao and G. Karypis. Evaluation of hierarchical clustering algorithms for document datasets. In *ACM Conf. Information & Knowledge Management (CIKM)*, 2002.

A Graph Labeling and Abstraction

Nodes and edges that are inserted into the behavior graph are augmented with additional labels that capture more information about the nature of the system calls and the dependencies between nodes. In the following, we describe this labeling in greater detail. For edges, a label stores the names of the input and output arguments, respectively, of the system calls that are connected through a data dependency. In case of a node, the label stores the system call name and some optional information that depends on the specific type of call.

As shown in Table 1, the additional information can correspond to the type of the resource (files, registry keys, ...) that a system call operates on as well as flags (such as mode or permission bits for file operations). Note that some information is only stored as comments; this information is ignored for the template generation and matching, but is saved for a human analyst who might want to examine a template.

Operations	File	Registry	Network
Label	Location, Type (Table 2), Access, Attributes, CreateDisposition	Key name, Value name	Port
Comment	File name		IP address

Table 1: Selected information for labels and comments.

One important additional piece of information stored for system calls that manipulate files and registry keys is the name of these files and keys. However, for these resource names, it is not desirable to use the actual string. The reason is that labels are taken into account during the matching process, and two nodes are considered the same only when their labels match. Thus, some type of abstraction is necessary for labels that represent resource names, otherwise, graphs become too specific.

In the case of files, the name string is split into three parts: the path representing the location of the file, the short name of the file and its extension. Table 2 shows how the paths, short names and extensions are mapped to several generic classes of location and type, that are then used for the file name label. Similarly, the registry key names are split into two parts: the location of the key and its short name. The location is first normalized using the standard registry abbreviations (`HKLM`, `HKU`, `HKCU`, `HKCR`). The short key name is then confronted to generic types (*number*, *path*, *url*). If the name does not comply with any format, but still shows a high number of similar close variations, a generic type *random* is attributed. Additional examples of this abstraction process can be observed in the examples of template of the next section.

B Template Examples

We manually examined C&C templates to determine whether they capture activity that a human analyst would consider malicious. We now present two examples that were automatically generated by JACKSTRAWS.

Figure 4 shows a template we extracted from bots that use a proprietary, binary protocol for communicat-

Location	File Path	Type	Extension
InWindowsDirectory\	\Windows\	IsExecutable	*.exe
InSystemDirectory\	\Windows\System*\	IsDynamicLibrary	*.dll
InDocumentDirectory\	\Documents and Settings\	IsDriver	*.sys, *.drv
InStartupDirectory\	\Documents and Settings*\Startup\	IsConfiguration	*.ini, *.cfg
InTemporaryDirectory\	\Documents and Settings*\Local Settings\Temp\	IsWebPage	*.htm, *.php, *.xml
InInternetDirectory\	\Documents and Settings*\Local Settings\Temporary Internet Files\	IsScript	*.js, *.vbs
InProgramDirectory\	\Program Files\	IsCookie	\Cookies*@*.txt
		IsDevice	\Device\
		IsNetworkDevice	\Device\AfdEndPoint

Table 2: File abstraction based on location and type.

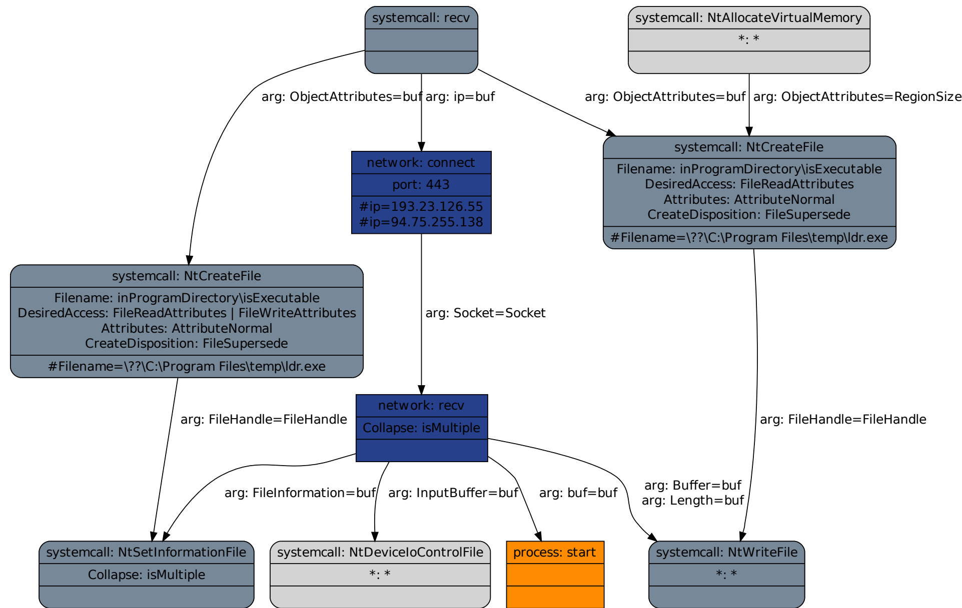


Figure 5: Template that describes the download and execute functionality of a bot: an executable file is created, its content is downloaded from the network, decoded, written to disk, its information is modified before being executed. In the *NtCreateFile* node, the file name *ldr.exe* is only mentioned as a comment. Comments help a human analyst when looking at a template, but they are ignored by the matching.

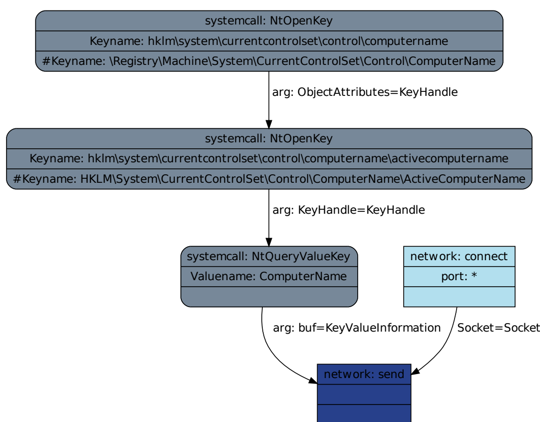


Figure 4: Template that describes leaking of sensitive data. Darker nodes constitute the template core, whereas lighter ones are optional.

ing with the C&C server. The behavior corresponds to some kind of *information leakage*: the samples query the registry for the computer name and send this information via the network to a server. We consider this a malicious activity, which is often used by bots to generate a unique identifier for an infected machine. In the network traffic itself this activity cannot be easily identified, since the samples use their own protocol.

As another example, consider the template shown in Figure 5. This template corresponds to the *download & execute* behavior, i.e., data is downloaded from the network, written to disk, and then executed. The template describes this specific behavior in a generic way.