Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries

Mario Polino[1], Andrea Scorti[1], Federico Maggi[1], and Stefano Zanero[1]

DEIB, Politecnico di Milano, Italy
{mario.polino,federico.maggi,stefano.zanero}@polimi.it
andrea.scorit@mail.polimi.it

Abstract When analyzing an untrusted binary, reverse engineers usually rely on ad-hoc collections of interesting dynamic patterns—known as behaviors in the malware-analysis community—and static patterns—known as signatures in the antivirus community. Such patterns are often part of the skill set of the analyst, sometimes implemented in manually-created post-processing scripts. It would be desirable to be able to automatically find such behaviors, present them to analysts, and create a systematic catalog of matching rules and relevant implementations. We propose JACKDAW, a system that finds interesting dynamic patterns, and ranks them to unveil potentially interesting behaviors. Then, it annotates them with static information, capturing the distinct implementations of each across different malware families. Finally, JACKDAW associates semantic information to the behaviors, so as to create a descriptive summary that helps the analysts in querying the catalog of behaviors by type. To do this, it leverages the dynamic information and an indexed Web-based knowledge databases.

We implement and demonstrate JACKDAW on the Win32 API (even if the technique can be generalized to any OS). On a dataset of 2,136 distinct binaries, including both malicious and benign libraries and executables, we compared the behaviors extracted automatically against a ground truth of 44 behaviors created manually by expert analysts. JACKDAW found 77.3% of them and was able to exclude spurious behaviors in 99.6% cases. We also discovered 466 novel behaviors, among which manual exploration and review by expert reverse engineers revealed interesting findings and confirmed the correctness of the semantic tagging.

1 Introduction

The increasing interest around reverse engineering complex legacy, untrusted or malicious binaries demands for automated tools that aid the analysts. Recent works such as Howard [36] or TIE [20] propose intelligent solutions to tackle some of the hard aspects of "reversing". We believe that research efforts in this direction are needed to turn reverse engineering from an art into a more structured engineering discipline with appropriate methodologies, tools and computer-supported processes.

Research Challenges. One of the main challenges of reverse engineering is achieving automation, in order to overcome the shortage of skilled analysts. A variety of static- and dynamic-analysis tools exist and are very useful to this end. Moreover, so-called "hybrid analysis" approaches can be used to balance their symmetric strengths and weaknesses [6, 10, 21, 34]. We believe that hybrid approaches can be pushed forward and leveraged to obtain better reverse engineering tools.

The core aim of hybrid analysis techniques is to help bridging the semantic gap between static and dynamic analysis, using as a pivot the concept of *behavior*, expressed in different ways and abstraction levels (e.g., groups of API or system calls, instruction sequences). Thus, the automatic identification of such behaviors is an interesting and challenging research problem with immediate and profund practical impact. Such an output could, for example, be used by a plugin for reverse-engineering tools to automatically highlight and annotate certain portions of the CFG to prioritize the analysis based on the information extracted from a large, collaborative back-end database, freeing up valuable analyst time to focus on novel, interesting behaviors. Such behaviors could then be fed back into the behavior database

Goals and Approach. We propose a practical approach to automatically extract behavior specifications. JACKDAW is based on the correlation of control- and data-flow information extracted from binaries both statically (after unpacking) and dynamically.

Under the realistic assumption that data-flow-dependent APIs or system calls are signs of strictly-related events, we can automatically recognize groups of relevant actions that could constitute a behavior, without any previous knowledge of that behavior. More precisely, if a sequence of API or system calls connected through data-flow dependency (which we call *sequence of dataflow-dependent API calls* for brevity) is recurrent within many binaries, it could constitute a meaningful and interesting behavior.

Obviously, frequency alone is not enough. We exploit the observation of previous work (e.g., [21]) that code reuse in malware is common, also across families that evolve independently. Thus, we leverage the availability of a large number of samples to find different, recurring implementations of such behaviors, which confirm their consistency. In a way, we are turning the abundance of malware variants against the adversaries.

System Overview JACKDAW first identifies candidate behaviors as groups of API functions. The groups are formed by means of dynamically extracted data-flow dependencies, and using a similarity criterion based on the Control Flow Graph (CFG). Then, JACKDAW builds a model of each group, which essentially is the list of the most frequent function calls contained. Finally, JACKDAW leverages a knowledge base (e.g., StackOverflow) to associate function names and semantic tags.

Impact. The difference between JACKDAW and previous works that employ clustering in malware analysis is clear: JACKDAW does *not* cluster the *binaries* in any way. Instead, it uses clustering techniques to *assist the discovery* of (relevant) behaviors automatically. Clustering executable binaries based on shared, known behaviors has been already done in the past [17, 33] and is not the focus of our work. In fact, our motivation is exactly the opposite, we do not assume any knowledge about behaviors. Thus, the goal is to find recurrent and correlated data-flow and CFG sub-graphs that could represent a behavior.

The output of our system (i.e., a list of high-level dynamic traces enriched with static information) could be used as input to other hybrid analysis systems that need behavior definitions to work, thus relieving the analyst from the burden of producing behavior specifications manually. It could also be used to build binary clustering or classification techniques. Finally, it could be used by a plugin for reverse-engineering tools in order to automatically annotate portions of the CFG that implement behaviors.

Evaluation Summary. We compared the behaviors extracted automatically by JACKDAW against a ground truth of 44 known behaviors constructed manually (and tediously) with the help of a malware analyst. Our results on real-world malicious and benign binaries indicate that JACKDAW finds up to 77.3% of the ground truth behaviors, and effectively "suggests" interesting new ones: We verified this by validating novel extracted behaviors with the help of a panel of malware analysts. In a similar fashion, we were able to validate the automated semantic tagging of behaviors. Moreover, when applied across binaries of distinct categories (e.g., malicious vs. benign), we show that JACKDAW is useful to lookup in the benign binaries (only) those behaviors constructed from malicious binaries, with high precision. Indeed, when used to query a behavior catalog constructed from malicious binaries, only 0.4% of such behaviors are found in benign binaries, showing that JACKDAW could be used to perform differential analysis and similar tasks. Finally, we show that JACKDAW can recognize (new) behaviors in binaries never seen before.

Contributions. In summary:

- We present an unsupervised approach to ease and systematize the task of reverse engineering by automatically extracting high-level behavior descriptions from a large dataset of binaries.
- We remove a time consuming manual step in hybrid static-dynamic analysis processes, namely the definition of high level behaviors.
- We propose an automatic algorithm to associate semantic tags to behaviors, allowing (inexperienced) analysts to understand their actions.

2 Binary Analysis and Reverse Engineering

Static and dynamic analysis techniques have symmetric pros and cons. The key advantage of static analysis is the good code coverage, whereas the main disadvantage is that it requires skills and time to understand the results. Moreover, these techniques suffer from compiler optimization, packing, obfuscation, polymorphism, and other code-transformation techniques applied both to goodware, for intellectual-property protection, and to malware, for evading static signatures. Dynamic-analysis techniques are based on tracking events at different abstraction levels (e.g., machine instructions, file system writes, network activity, auto-update capabilities, registry actions) while a binary executes. The relevant events are obtained in various ways (e.g., API hooking in user or kernel mode, custom kernel). Symmetrically to static analysis, dynamic analysis requires significantly less skills and effort to be understood and is resilient to code transformations. The main limitation is that we can only analyze the pieces of code that are actually executed during the analysis. Therefore, some features remain hidden, either by chance or intentionally (e.g., evasion). Code coverage can be increased with proper code-stimulation techniques, at the price of an increased complexity, and by relying on hardware-level introspection. As shown by a recent quantitative analysis [40], a combination of static and dynamic analysis, creating so-called *hybrid* approaches, is the key to achieve the best recall and precision.

We observe that previous work revolve around the concept of *behavior* [16, 26], which is leveraged as the bridge between static an dynamic techniques. This concept has been used for various purposes, ranging from classification to analysis [21] and detection. In a general meaning, a behavior is a set of events—possibly along with arguments and types (see [20, 36])—observed during dynamic analysis. The classic example is a trace of system or API calls. Current approaches require that the analyst defines behaviors manually To take reverse engineering and malware analysis one step further, our first goal is to generate (candidate) behaviors in a fully automatic way. More precisely, we want to find groups of API calls that *could be* the building blocks of a higher level action, from the results of hybrid analysis. Our second goal is to reduce the gap between the API functions used to describe a behavior and their semantic meaning (i.e., what the analyst wants to understand). In other words, we want to assign a name to a behavior, and thus complete the bottom-up recognition of the top-level behavioral description in [26]. Then, we want to attach static information (e.g., basic blocks) to the extracted behaviors.

Our starting point are the data dependencies, identified through dynamic data-flow analysis (DFA). In short, DFA allows to "track" data in memory by following copy or other manipulation operations. For instance, this means that the result of an operation receives the union of labels of the operation's arguments. Although in principle any DFA technique could be used, we base our implementation on value-based DFA, by connecting the return values (and output arguments) of one API call with the input arguments of subsequent calls, creating sets of connected calls. These sets are the initial candidates to mine behavior specifications. The challenge, however, is that these sets can contain API calls that are irrelevant to defining a behavior. We want to find smaller groups of API calls that are the "core" of these sets and that frequently occur together, in a way automating the process of identifying bottom-up the "lower level" of the hierarchy explained in [26].

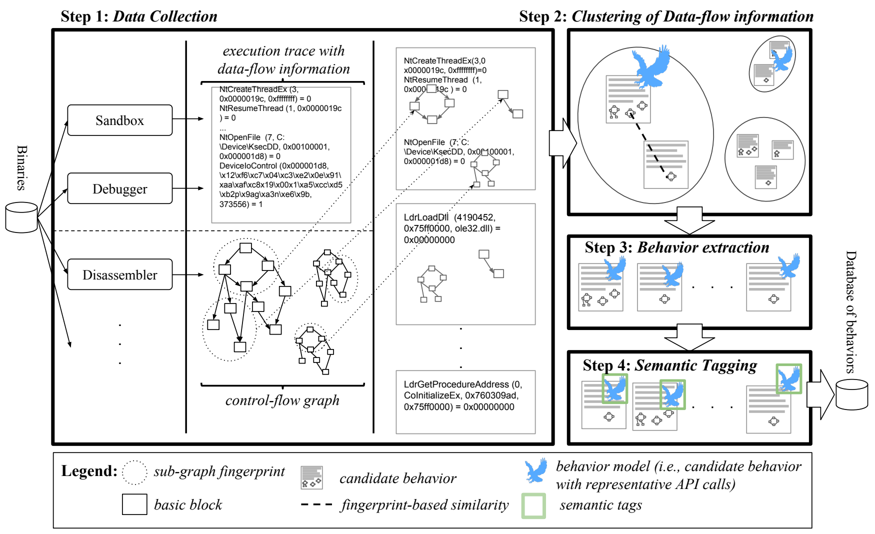
3 System Details

As summarized in Fig. 1, JACKDAW comprises four steps. **Step 1 (Data Collection)** collects and pre-processes static and dynamic information, **Step 2 (Clustering of Data-flow Information)** groups the sequences of dataflow-dependent API calls that have the same CFG fingerprints, and in **Step 3 (Behavior Extraction)**, the clusters are modeled by means of the representative API calls found in the sequences of dataflow-dependent API calls. **Step 4 (Semantic Tagging)** attaches meaningful tags to each extracted behavior.

3.1 Step 1: Data Collection

We let each binary run in a monitored environment, this preliminary execution step also unpacks packed code, if any, by letting the executable run and then dumping the process memory. Clearly, a solution to the generic and challenging problem of packing is out of the scope of this work. During execution we collect the set of sequences of dataflow-dependent API calls and map any data dependencies to the code where the API function is called. Here, we represent the *static code* by means of sub-graphs of the CFG (built from the memory dump), which we call *fingerprints*.

Introspection and DFA. In our implementation we use an introspection technique proven to work well even in case of malicious binaries [7, 27, 31]. During tracing, we attach labels to interesting data flows. The flow sources are the API calls (with their return values). We propagate these labels whenever they are copied or otherwise manipulated. This is essentially taint analysis, although we prefer to use the term data-flow analysis, which is more generic. Indeed, if type information is available, our technique can be


Step 1: Data Collection
Step 2: Clustering of Data-flow information
Binaries
Sandbox
Debugger
Disassembler
execution trace with data-flow information
control-flow graph
Step 3: Behavior extraction
Step 4: Semantic Tagging
Database of behaviors
Legend:
sub-graph fingerprint
candidate behavior
behavior model (i.e., candidate behavior with representative API calls)
basic block
fingerprint-based similarity
semantic tags

```
NtCreateThreadEx (3,
0x0000019c, 0xffffffff) = 0
NtResumeThread (1, 0x0000019c
) = 0
...
NtOpenFile  (7, C:
\Device\KsecDD, 0x00100001,
0x000001d8) = 0
DeviceIoControl (0x000001d8,
\x12\xf6\xc7\x04\xc3\xe2\x0e\x91\
xaa\xaf\xc8x19\x00x1\xa5\xcc\xd5
\xb2p\x9ag\xa3n\xe6\x9b,
373556) = 1
```

```
NtCreateThreadEx(3,0
x0000019c, 0xffffffff)=0
NtResumeThread  (1,
0x0000019c ) = 0

NtOpenFile  (7, C:
\Device\KsecDD, 0x00100001,
0x000001d8) = 0
```

```
LdrLoadDll  (4190452,
0x75ff0000, ole32.dll) =
0x00000000
```

```
LdrGetProcedureAddress (0,
CoInitializeEx, 0x760309ad,
0x75ff0000) = 0x00000000
```

Figure 1. Step 1: Data Collection combines static information (CFG) and dynamic information (execution traces). **Step 2: Clustering of Data-flow Information** groups the sequences of dataflow-dependent API calls that have the same CFG fingerprints. In **Step 3: Behavior Extraction**, the clusters are modeled by means of the representative API calls found in the sequences of dataflow-dependent API calls. **Step 4: Semantic tagging**, semantic tags are attached for each model.

easily extended. In practice, for each API function called, we extract the parameters' name and actual value at the moment of invocation. Then, we use data-flow information to connect the return values (and out-arguments) of one API call with the in-arguments of subsequent calls.

Static Information. We also map each sequence of dataflow-dependent API calls to the code that implements it. To represent (and compare) code blocks in a way that is resilient to recompilation in different contexts, we use the notion of *fingerprints*, which are sub-graphs of size k (in our work, as in previous and recent work, we use $k = 10$) of the colored CFG obtained from the unpacked binary code. The unpacking step is obviously optional. The CFG is colored according to the types of instructions contained in each basic block. Each sub-graph can be conveniently represented and efficiently matched using a hash. This technique is proven to be resilient against polymorphism [19] and was successfully used in recent samples by Comparetti et al. in [6]. Finally, we normalize the API function names (e.g, removing *'A','W','Ex'* suffixes referring to different versions of the same function).

Behavior (definition). At this point, we can define a *behavior* as a sequence of API function names and parameters, and the respective static fingerprints used to identify its implementation on the CFG.

3.2 Step 2: Clustering of Data-flow Information

The input of this phase is a set of sequences of dataflow-dependent API calls, enriched with static information. The goal is to group sequences of dataflow-dependent API calls

```
Input: dataflow_set, clusterset=$\{c_1...c_l\}$
for all $t \in dataflow\_set$ do
    for $i \in \{1...l\}$ do
        $s_i \leftarrow J(t, c_i)$ {Jaccard similarity computation}
    end for
    $s_{i^\star} \leftarrow argmax(s_i)$
    if $s_{i^\star} > u$ then
        $c_{i^\star} \leftarrow c_{i^\star} \cup \{t\}$
    else
        new $c_{l+1}$
        $c_{l+1} \leftarrow \{t\}$
        $clusterset \leftarrow clusterset \cup c_{l+1}$
        $l \leftarrow l+1$
    end if
end for
```

Figure 2. Clustering algorithm based on ECM: $s_{i^\star}$ is the maximum Jaccard similarity between t and all the sequences of dataflow-dependent API calls in the i-th cluster c_i.

by their similarity. The requirements are a simple, fast clustering algorithm to keep the analysis time under control. The algorithm must be one-pass and distance-based, where clusters can evolve in an on-line mode as new samples come in. To this end, we customized the ECM algorithm [38] as explained in the reminder of this section.

Clustering Algorithm. As detailed in Fig. 2, we associate each item to the cluster with the highest (Jaccard) similarity. Recall that the items to be clustered are the sequences of dataflow-dependent API calls, each represented by their set of fingerprints. The original ECM algorithm uses average linkage to compute the distance. However, the concept of "average set of fingerprints" is meaningless in our domain. Thus, we use single linkage. For this reason, if the distance to the closest cluster $s_{i^\star}$ is higher than a threshold (empirically set to $u = 0.75$, as justified in §4.2), the item is considered the first item of a new cluster.

Distance Function. Conceptually, two sequences of dataflow-dependent API calls are very similar if they share large parts of code fingerprints. Therefore, we use the fingerprints—indexed with hashes—to calculate a similarity score between each pair of sequences of dataflow-dependent API calls.

After having considered several distance metrics suitable for sets (proposed in [4]), we ended up comparing the *dice coefficient*, the *Jaccard similarity* and the *overlap coefficient*. Through a set of experiments, we concluded that Jaccard similarity works best for purposes: $J(A,B) = |A \cap B|/|A \cup B|$, where A and B are two sets of fingerprints.

Since our distance metric is based on the Jaccard index, alternative approaches could be used (e.g., locality-sensitive hashing). However, we consider exploring such alternatives an orthogonal aspect of our contribution, beyond demonstrating our idea.

We run a pilot experiment on a distance function based on the API calls that two sequences of dataflow-dependent API calls have in common. However, this approach would yield very sparse clustering, too biased by dormant code. Indeed, we show that different sets of APIs are found in similar portions of code (i.e., implementing the very same behavior).

3.3 Step 3: Behavior Extraction

Each cluster is now a potential candidate behavior, of which we know that all elements share similar code. The goal of this step is to create a succint cluster model that represents the sequences of dataflow-dependent API calls that it contains. For this, we need to find a small set of API calls that characterize each cluster. To this end, we propose an heuristic based on the frequency of each API call, which searches for sequences of dataflow-dependent API calls that have sub-sequences of API calls in common. Our hypothesis, validated by our experiments, is that they will also share (parts of) the implementation code. As a result, since each cluster carries the respective fingerprints, we have obtained a *dictionary of behaviors* (i.e., set of APIs) with several implementations (i.e., CFG fingerprints). In addition to the fingerprint information that is useful for fast indexing, we store the static information associated to each behavior as a set of offsets that identify the code into the binary. Therefore, this dictionary can be used to statically match behaviors, both in new, unseen binaries, or in binaries where such behaviors are implemented but "dormant" [6]. Notably, the ability to produce both a static and dynamic description of behaviors is useful because in this way we are not bound to dynamic analysis to identify behavior in samples.

Extraction Logic. To find the set of APIs that will build a cluster model we consider an API as part of a behavior if it appears frequently within the same cluster. This means that, given the same portions of code (i.e., CFG fingerprints of a cluster), the most frequent APIs are those that are manifested during dynamic analysis. An API is representative of a cluster, and thus part of its model, if it appears more frequently than a threshold f in the cluster. We found $f = 0.75$ to be a good value empirically, as justified by our experiments. A conservative choice would be $f = 1.00$, which would mean that we use as a descriptor only APIs that appear in all the sequences of dataflow-dependent API calls of the cluster. This, however, is brittle and decreases the amounts of behaviors found (as will be evident in §4.2): Indeed, it does not take into account dormant behaviors [6], or the possibility that a behavior contains multiple branches and thus alternative portions of code.

At this point, as shown in Fig. 1 (right), we merge the behaviors that have the same API function names, thus enriching the set of fingerprints (i.e., code fragments) associated to each behavior. Two clusters are merged if their have the same model (i.e., same set of representative APIs). Recall that although the cluster modeling is done at behavioral level, the clustering is obtained based on a distance function calculated on the CFGs (i.e., static code level). Therefore, merging these models will produce clusters that contains several sets of these fingerprints, each representing a specific implementation of the behavior. Consequently, distinct clusters could yield the very same model.

As a result, we obtain a linked graph (excerpt in Fig. 10). This creates our catalog of behaviors. In this graph we show the inclusion relationships, to show which behaviors depend on which behaviors, pretty much like the behavior graphs produced by [26].

Type Information. If type information is available from **Step 1**, we take into account also the arguments of the representative APIs. To this end, we ported to Win32 the well-known, robust models implemented in [24, 29] for the most common data types of Linux system calls. We focused on the four most common types of parameters in our proof of concept implementation: strings, tokens, IP addresses and transport protocols. The first two are precisely the same present in [24], while the latter two are just specializations of the token model.

3.4 Step 4: Semantic Tagging

The goal is now to tag candidate behaviors with human-readable semantic descriptions. The rationale is that each API call has a role in building the overall behavior. The final result is a set of candidate behaviors and dictionaries of their implementations, and tagged with keywords that can help the analyst in determining their semantics.

Sources of Semantic Tags. We first explored the official MSDN documentation. However, it considers the API functions when used alone, not in various combinations. Therefore, we exploited the abundant, structured information available nowadays in community-driven websites. As a proof of concept, we obtained data from StackOverflow, a popular community-based Q&A website extensively used by programmers. Questions about programming problem often include code snippets, and are always tagged as enforced by the site. In principle, any of such knowledge databases could be used, including custom ones built by the analyst over time, which makes JACKDAW fairly flexible.

Tagging Algorithm. For each element of the power set of the set API function names and parameters of each behavior, we search our dump of StackOverflow for questions that contain that element and extract title, body and tags. For example, given a behavior that contains `Connect Port 25`, for each StackOverflow result[1] we extract the title (e.g., "Send mail through gmail SMTP server using Win API"), body and tags (e.g., "winapi", "smtp", "gmail"). For each post, we compute Score$(post)$ from two configurable lists:

- **Interesting tags list**: we add $+1$ for each word of this list contained in a post; this (extensible) list currently contains `^c$`, `c\+\+`, `c\#`, `win` in order to lookup posts strictly related to Windows APIs.
- **Trifling tags list**: we add -1 for each word of this list contained in a post; this (extensible) list currently contains words such as `php`, `python`, and other language names, which suggest that the post is related to a specific programming language or context, not to the Win API.

These lists are an important customization aspect of a reverse-engineering product based on our technique: Indeed, the analyst should be able to tailor her lists based on the focus, which can obviously change.

Any post with a positive overall score is marked as relevant. We then weight the resulting relevant tags and posts with

$$\text{Score}(tag, post) = \frac{\text{Score}(post)}{N} \cdot \text{Found}(post, tag)$$

where Found$(post, tag) \in 0, 1$ is 1 only if the $post$ contains the given tag, N is the number of relevant posts. With this, we can calculate a vote for each tag as

$$\text{Vote}(tag) = \sum_{post \in \text{All posts}} \text{Score}(post)$$

which we use to build a ranked list of tags. The human analyst can choose how many suggestions (s)he wishes to see for each behavior. Thanks to our experimental results

[1] http://stackoverflow.com/questions/3281260/send-mail-through-gmail-smtp-server-using-win-api

(§4.4) we discuss the quality of the suggestions, and show that applicable clues to the behavior semantic meaning show up very early in the list. Our experiment shows that tags are already a useful means to create a succinct description of a behavior. Moreover, indexing and querying tags is space and time efficient. Therefore, we leave more complex techniques borrowed from natural language processing field as future improvements, as discussed in §5.

4 Experimental Evaluation

Our main goal is to validate the behaviors produced by JACKDAW. We apply it to both malicious and benign binaries. We first show that behaviors extracted from a large corpus of real-world malware samples from different families are consistent with those that an analyst would have found via manual analysis. In addition, we show that these behaviors are found on new malware families, never seen before by our system. Secondly, we show that the behavior catalog so produced does not include spurious behaviors. In fact, we show that the behaviors extracted from malicious binaries are not found in benign binaries. Clearly, some behaviors can be in common between different classes of binaries, but their implementation usually differs. JACKDAW carries such (static) information with each behavior, and thus makes this type of reasoning feasible. As a side result, our findings suggest that the behaviors discovered by JACKDAW may be used to some extent to classify binaries on a per-behavior basis, although this should be done cautiously because behaviors could be shared. Last, we conduct a survey from which we obtained 71 responses from a pool of expert reverse engineers that provided positive feedback on the usefulness of the behaviors produced by JACKDAW.

We hosted JACKDAW on an Intel Core i7 CPU Q 720 @ 1.60GHz, with 4GB of RAM, running Linux. We obtained access to the Anubis [1] sandbox for **Step 1**.

4.1 Dataset and Ground Truth

We collected 1,272 (about 10GB of data flow traces) samples belonging to 17 malware families (Banload, Cycbot, Dapato, Gamarue, Generic Downloader, Generic Trojan, Graftor, Kelihos, Llac, OnlineGames, ZangoHotbar, and ZeuS). Additionally, for the experiment described in §4.4 we used a dataset comprising 864 distinct benign binaries (e.g., PE32 executables and DLL libraries) extracted randomly from the `Windows/` sub-directory of a clean computer.

To validate our system we needed *known* behaviors. For this, we manually reverse engineered one sample from each family and, in addition, we manually inspected the sequences of dataflow-dependent API calls, extracted through **Step 1**, from randomly picked samples. With the help of a malware analyst, we ended up creating 44 ground truth behaviors, subsequently validated by other two distinct reverse engineers. We included behaviors in the following categories: network activity, download & execute, file harvesting, history harvesting, disabling task manager, browser hijacking, disabling an AV, autorun, disabling Windows firewall, unpacking. In §4.3 we use these behaviors as a reference to tune our parameters, in §4.4 we use these behaviors as an oracle to determine if JACKDAW would be able to find them automatically.

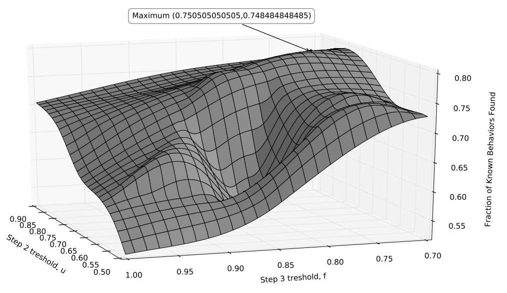


Figure 3. Selection of the parameters used in **Step 2** and **Step 3** by optimization of the number of behaviors found in a labeled portion of the dataset.

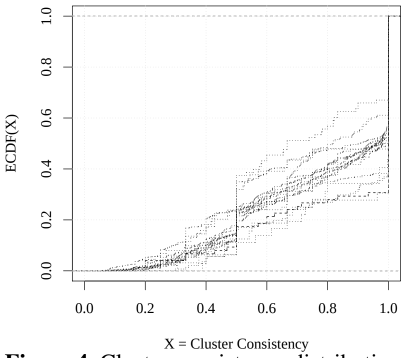


Figure 4. Cluster consistency distribution of all families (one line per family).

4.2 Parameter Estimation

In **Step 2** and **3** (§3.2 and 3.1) we introduced two parameters. One of these parameters is the threshold u used by the clustering algorithm to decide to split an incoming item and form a new cluster around it. High values of u yield many clusters (i.e., candidate behaviors), with representative sets that would have a large number of API calls, which then the merging step would hardly be able to merge again.

A second parameter, f, determines how frequent an API needs to be in a cluster to be considered a representative. A conservative choice would be $f = 1.00$, which would mean that only APIs that appear in all of the sequences of dataflow-dependent API calls of the cluster would be a descriptor. This would create very small descriptions, and decrease the numbers of behaviors found: Some clusters may not have APIs that appear in all sets at all; some others may been implemented using different APIs. On the other hand a lax choice would generate "representative sets" that would contain spurious APIs, and thus not match any real behavior.

Since their effects interfere, it is necessary to jointly estimate these parameters. To do so, we exploit the availability of a ground truth of known, manually labeled behaviors. In Fig. 3 we plot, on the x and y axes, the two parameters, respectively, and on z the fraction of the ground truth behaviors extracted by JACKDAW with that combination of parameters. This surface has a global maximum in $f = 0.75$ and $u = 0.75$. It should be noted that the sensitivity with respect to this choice is not high (because of the smoothness of the surface), so we can use these parameter values safely.

4.3 Clustering Validation (Step 2)

The first set of experiments is targeted to validating the sequence of steps we designed and the assumptions we took for each step.

Similarity Metric Selection In order to select the best similarity metric for sequences of dataflow-dependent API calls, we want our clustering to be able to group sequences of dataflow-dependent API calls that belong to the same behavior together. We used our 44 ground-truth behaviors as a reference exploring metrics chosen in [4] .

In Table 1, sequences of dataflow-dependent API calls that are similar according to each metric and that belong to the same reference behaviors are marked in white (correct),

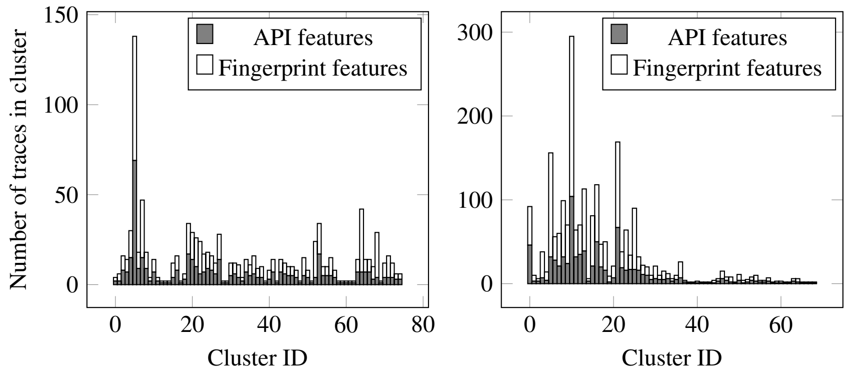


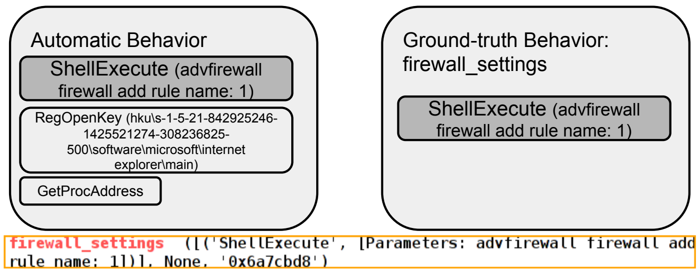
Figure 5. Consistency of clustering (`Banload` left, `Graftor` right). Each cluster is plotted in a separate bar. In white we plot the code clusters, and in gray we superimpose the largest sub-cluster according to the APIs. On the y-axis there is the number of sequences of dataflow-dependent API calls contained in the cluster.

whereas the sequences of dataflow-dependent API calls that are similar only according to the metric but that belong to different behaviors are marked in gray (incorrect). To minimize such misclassification, we can use either the Dice or Jaccard distance with similar performance, whereas the overlap coefficient consistently perform worse. According to these results, we chose the Jaccard similarity for all of our next experiments, which is also fast to compute.

Consistency The aim of this experiment is to verify whether the clusters obtained using fingerprints are consistent with their representative API functions. In other words, we want to verify our assumption that if we cluster samples using only static features (i.e., CFG fingerprints), and then cluster the same samples using only dynamic feature (i.e., API calls), we end up obtaining consistent clusters (i.e., basically the same clusters). To verify this, we proceeded as follows. Operating on each cluster, we applied the same clustering algorithm, first on (sets of) fingerprints then on (sets of) API function names. In an optimal case, such algorithm should generate the same clustering. To visually represent this, in the examples on Fig. 5 we plot each cluster in a separate bar (white), and we superimpose the largest sub-cluster in that cluster according to the APIs (gray). On the *y*-axis there is the number of sequences of dataflow-dependent API calls in the cluster.

Table 1. Results of similarity metric selection (**Step 2**).

Family	Dice Coefficent		Jaccard Similarity		Overlap Similarity	
API sequence →	#	%	#	%	#	%
Banload	63	75	63	78	87	51
	21	25	18	22	82	49
Dapato	97	86	97	86	6	03
	16	14	16	14	191	97
Gamarue	16,411	53	15,142	53	13,634	34
	14,416	47	13,583	47	25,940	66
Llac	412	90	412	91	209	86
	43	10	41	09	34	14
Gen.Downloader	8,676	64	8,211	72	12,095	29
	4,807	36	3,216	28	29,193	71


Automatic Behavior
ShellExecute (advfirewall firewall add rule name: 1)
RegOpenKey (hku\s-1-5-21-842925246-1425521274-308236825-500\software\microsoft\internet explorer\main)
GetProcAddress
Ground-truth Behavior: firewall_settings
ShellExecute (advfirewall firewall add rule name: 1)

```
firewall_settings  ([('ShellExecute', [Parameters: advfirewall firewall add
rule name: 1])], None, '0x6a7cbd8')
```

Figure 6. Comparison of handwritten rule for a `Firewall settings` behavior and of automatically extracted specification (on the top left).

Fig. 5 shows the results for `Banload` and `Graftor`, respectively. The results are good, because in almost all clusters the largest sub-cluster contains well above 50% of the sequences of dataflow-dependent API calls, and in some cases up to 100%. This means that our clusters are eligible for a further API functions extraction. The results on this experiment on all the malicious binary families are very similar, as summarized in Fig. 4 by means of the empirical cumulative distribution function $CDF(X)$, where X is the percentage of containment (gray bars): Most of the values are high (notice the significant density around 1.00), showing good consistency overall. Thus, we conclude that our hypothesis is empirically correct. Therefore, we can safely use the static fingerprints (i.e., implementation of a behavior) as features of a behavior.

4.4 Behavior Evaluation (Step 3)

We assess whether the behaviors extracted by our approach (1) are meaningful, (2) can be recognized in unknown binaries, and (3) correctly tagged.

Evaluating the results of JACKDAW is challenging, because it produces novel knowledge (i.e., new behaviors) using unsupervised methods. The reason is that we could not possibly have (found) all these behaviors in our ground truth, and they would require an analyst to manually verify if they are reasonable, and if tags are consistent with API functions contained therein. Having considered these aspects, in addition to compare our behaviors against the 44 behaviors defined by an expert we perform a one-off cross-validation of our behaviors.

Comparison Against Ground Truth We compared the behaviors extracted automatically by JACKDAW, against our ground truth of 44 behaviors built manually. This is of course not a comprehensive test of effectiveness, but rather a test of quality and precision. Overall, in about half hour JACKDAW processed our dataset was able to find, among the extracted behaviors, 34 (77.3%) of the 44 ground-truth behaviors, which would have required days of tedious reverse engineering.

From our dataset of 1,272 malicious binaries we automatically generated 607 distinct behaviors with 2 to 3 average API functions each. Of these, 172 (28.3%) were known to the analyst. Notably, the behaviors that we extracted are even more informative than those extracted manually: They contain more contextual information, as depicted in Fig. 6. In some cases JACKDAW reports more details that the analyst was able to specify

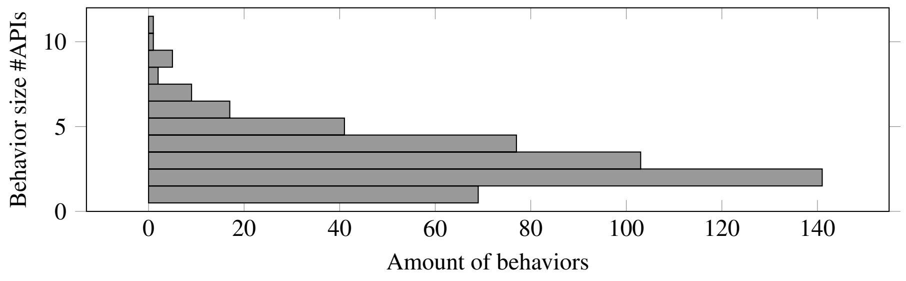


Figure 7. Number and size (#APIs) of the new behaviors in the catalog (i.e., those not matching the behaviors specified manually by our reverse engineers).

manually (e.g., the `RegOpenKey` and `GetProc` address in Fig. 6). This useful information is part of the behavior.

Remarkably, by a manual analysis of the 435 remainder behaviors[2],which size is summarized in Fig. 7, we were able to find interesting, unknown behaviors. From this experiment we can conclude that JACKDAW is able to find, in our database, the ground-truth behaviors.

Matching Behaviors in Unknown Binaries Besides proving that JACKDAW extracts behaviors that match those specified by expert reverse engineers, we show that the behaviors and their implementations that JACKDAW builds are useful to analyze previously unseen binaries. To do so, we create a catalog of behaviors on the malware dataset, excluding one family at a time. Then, we draw random binaries from the excluded family and verify whether they contain the behaviors from our catalog. Note that the family labels are needed only for validating the results, not for our system to work.

Fig. 8 shows the fraction (in $[0,1]$) of behaviors found in each family. The high values demonstrate that JACKDAW is able to automatically construct behaviors that have high recall in other binaries. Thus, a reverse engineer equipped with a dataset such as ours to run JACKDAW on, would have high chances to find behaviors in future instances of malicious binaries. As a side result, not only this is consistent with previous work [21], which demonstrated code reuse within the same malware family, but it also shows that malware developers tend to include similar behaviors across different families.

Behaviors Matching Across Classes of Binaries We show that the behaviors catalog constructed on a dataset of a given class of binaries (i.e., malicious, in our case) does not include spurious behaviors. Spurious behaviors are essentially "noise" for the reverse engineer that want to focus on behaviors typically found in malicious programs. Notably, the results of this experiment show that JACKDAW can be used to "subtract" such noise (e.g., removing instances of behaviors extracted by benign binaries from the behaviors extracted by malicious binaries). The reason is because these behaviors describe code that is typically found in benign programs. To this end, we picked 864 Windows portable executables and libraries chosen at random from a real system and we used the (static) fingerprints extracted from such benign executables to query our behavior catalog. We searched each of the 607 behavior against every 864 benign file, obtaining 524,448 comparisons. Of these comparisons, only 2169 (0.4%) were matching. Thus, only a

[2] `https://gist.github.com/anonymous/6129d822af1bf299ca8a`

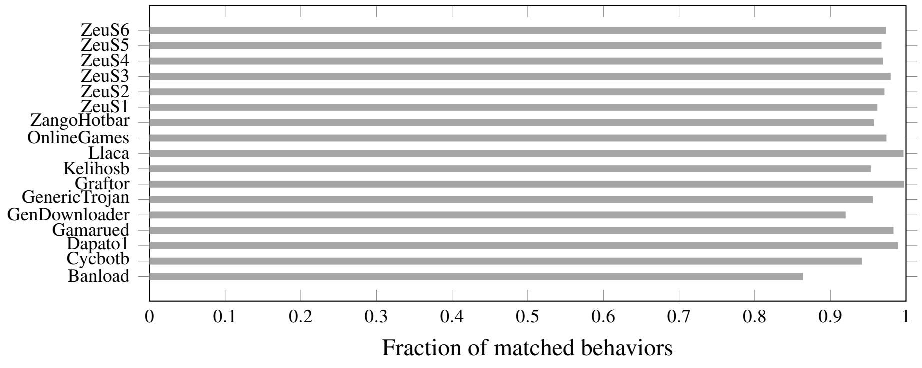


Figure 8. Matching behaviors in unknown malware (§4.4). Each bar shows the fraction of behaviors found in each malware family, obviously we deliberately excluded the samples of that family to build the behavior catalog used for matching.

```
ShellExecute(File: netsh, Parameters:
      advfirewall firewall add rule
      name)

RegOpenKey(hku\s
      -1-5-21-842925246-1425521274...
\software\microsoft\internet explorer\
     main)

GetProcAddress
```

Position	Tag (hint)	Score
1	netsh	71
2	registry	67
3	getprocaddress	49
4	dll	47
5	firewall	40
6	loadlibrary	19
7	installer	11
8	networking	10
9	wcf	8.1
10	mfc	8

Figure 9. Automatically extracted behavior (a change in firewall settings), and ranking of tags.

minimal fraction of behaviors from our malicious binaries were found in a benign dataset. This proves that the behaviors that we have extracted from malicious binaries are very useful for a reverse engineering to focus her analysis.

Interestingly, in those 2,169 matching comparisons we were able to find instances of behaviors that JACKDAW extracted from malicious binaries. More precisely, the downloading and execution of an EXE file was found in a malicious binary, and shared 88.46% of the code of a behavior found in the Adobe Updater. Indeed, the download-and-execute behavior is typical of benign software too.

Quality of Behavior Tagging To evaluate the quality of behavior tagging, we took 300 extracted behaviors, tagged them with **Step 4** (§3.4), and chose the first 40 tags from our ranking and manually analyzed the results. Let us discuss emblematic examples.

Fig. 9 shows what, at first sight, is a behavior that adds a rule to a firewall—as evident by looking at the parameter of `ShellExecute`. Interestingly, within the first 5 tags there are `netsh` and `firewall`. The tag `netsh` is the command name of the NetShell utility, which is used as a scripting interface for monitoring and configuring Microsoft Windows. These tags are very close to what the behavior does.

Survey on Behaviors Quality We asked expert reverse engineers to assign a score to randomly chosen behaviors. The scores were: "*correct*", if they understand the behavior and find it useful; "*complex*", if the behavior contains many APIs and should probably be split in sub-behaviors; "*no-sense*", if they think that the model of the behavior is wrong or useless. We obtained 71 answers: 67.71% (48) correct, 16.90% (12) complex, and 15.49% (11) no-sense. These results are promising and show that the approach on which JACKDAW is based can lead to better reverse-engineering programs that suggest interesting behaviors for the analyst. The low fraction of negative feedback that we obtained from the experts is an indication that not all the extracted behaviors are useful, which is expected from a fully automated system that requires no prior knowledge.

5 Limitations and Future Work

Despite the good results that we obtained, JACKDAW has some limitations.

First, our system shares a common limitation with all VM-based dynamic analysis tools: some binaries adopt anti-debugging or anti-virtualization techniques. This problem was first addressed in [14], which raised the problem of malware capable of fingerprinting honeypots and similar environments. It has been further shown that creating a VM or sandbox which cannot be distinguished from a real system is impractical or impossible [13]. In 2009, according to [2] 0.3 to 12.5 percent of the samples submitted to Anubis were able to detect the sandbox and refuse to run. In [22] it is shown that evasive malware was being actively developed and distributed in 2011. While overcoming this limitation is beyond the scope of our work, we can note that the problem of evasive malware is well mitigated by collecting data outside the malware execution scope. Current research focuses on using virtual machine introspection [9, 12, 30] instead of kernel- or user-space hooking, or work on the bare metal [8, 18, 39] so as to leave the ring 0 (and above) unaltered, thus limiting the malware environment fingerprinting possibilities.

A second, evident issue is that we need to unpack malware for our static analysis approach to work. We use a pragmatic approach to deal with packing: We let the executable unpack, de-obfuscate and run for at least 15 minutes and work on the memory snapshot. Defeating packing is not the focus of our work, and any of the more advanced unpacking techniques in the literature [15, 25, 34] can be used to augment our prototype.

Another limitation is the simple technique that we use to find the set of APIs that characterize a cluster. It works reasonably well, but we are considering a more complex heuristic that uses propositional logic to find whether there are (sequences of) API calls that perform similar actions but have different names and can be substituted for each other. An excerpt of an interesting case is depicted in Table 2. The analyst can easily spot that there is a rule hidden in this example: $Z = \texttt{OpenProcess} \wedge \texttt{EnumProcess} \vee \texttt{CreateTollhelp32Snapshot} \wedge \texttt{Process32First} \wedge \texttt{Process32Next}$. More formally, we want to extract and-or rules in the form Behavior $= \bigvee_i \bigwedge_j API_{i,j}$, where $API_{i,j} \equiv API_{i,j'}$ for all $j \neq j'$, for each i. The symbol $\equiv$ indicates that two API calls can be considered equivalent because, in all the sequences of all the clusters examined, they appear always together. On the one hand, being able to extract such rules would increase the generality of the extracted behaviors by composition of two existing behaviors. On the other hand, some preliminary tests suggest that the added complexity does not pay off in terms of effectiveness. This is due to the fact that to construct these propositional-logic rules we need an exhaustive search in the power set of the API function calls that are within each cluster, which does not scale in principle.

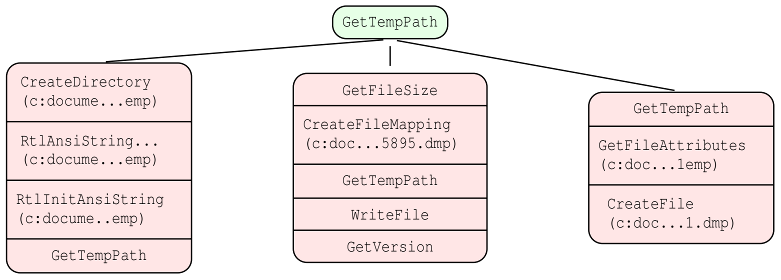

GetTempPath
CreateDirectory (c:docume...emp)
RtlAnsiString... (c:docume...emp)
RtlInitAnsiString (c:docume..emp)
GetTempPath
GetFileSize
CreateFileMapping (c:doc...5895.dmp)
GetTempPath
WriteFile
GetVersion
GetTempPath
GetFileAttributes (c:doc...1emp)
CreateFile (c:doc...1.dmp)

Figure 10. Example of behavior. The `GetTempPath` API calls alone are rather useless: Instead, behaviors connected to it highlight filesystem-related actions that are useful to the analyst.

Another future extension is to extract more generic behaviors by leveraging a concept that we call "behavior graph", a direct acyclic graph (DAG) where each node represents a behavior, and an edge exist between nodes if all of the APIs of a node (source) are included in the APIs of the other (destination). Nodes without incoming edges represent the most "general" behaviors. In many cases, they are composed by a single API, so they are so general that they do not really give useful information per se: we show this in Fig. 10 which magnifies a small portion of the graph to make it readable. However, by looking at behaviors "closely connected" to general behaviors (green nodes) we can see a broad classification of the behaviors in clusters such as "networking", registry, file operations, etc. Each behavior can belong to one or more of such groups. An interesting consequence that we are currently exploring is that it is possible to use this topological property to actually generate "missing" generalized behaviors.

6 Related Work

Dynamic Binary Analysis. Dynamic analysis approaches are based on sandboxes or otherwise instrumented environments. Some notable sandboxes are Anubis[3], to which we obtained access for our work, TEMU [37], CWSandbox[4], and Cuckoo[5]. Once the interesting events are collected, various post-processing techniques can be applied, for example to discover if the program is performing malicious actions. A recent, relevant work based on Anubis is [1], which extracts dynamic features and uses them to cluster similar malware samples together. We recall that this is the key difference with our work:

[3] `http://anubis.iseclab.org` [4] `http://www.cwsandbox.org`

[5] `http://cuckoosandbox.org`

Table 2. Each row shows the API calls found in a sequence of dataflow-dependent API calls. The names of the functions are split for space reasons.

Trace	**API calls (excerpt)**				
	Open-Process	Enum-Processes	CreateTool-help32-Snapshot	Process32-First	Process32-Next
1	☑	☑	☐	☐	☐
5	☑	☑	☐	☐	☐
6	☐	☐	☑	☑	☑
8	☐	☐	☑	☑	☑

They focus on finding groups of similar binaries, whereas we find relevant behaviors through clustering based on both dynamic and static features. Their aim is classification of malware, whereas ours is to use behaviors to aid reverse engineering.

Static Binary Analysis. Classic static analysis approaches on binary executables are based on disassembling the binary to obtain a higher-level representation based on the CFG, which has been shown to be abstract enough and resistant to obfuscation [3,19]. We use the CFG fingerprints defined in [19] as a mean to find recurring code across malware variants. In this specific part, our approach is similar to [5], which however focuses on detecting variants, whereas we provide a way to extract behaviors automatically, map them onto code, and find that code across variants.

More recent works such as [35] strive to push the abstraction further by leveraging the knowledge of the execution machine to obtain a de-compiled source code, which in principle carries more semantic than plain assembly code.

We focus specifically on explaining the reasons why an executable is malicious, by extracting the malicious behaviors and tag them semantically. Despite the good amount of research in this area, static analysis require manual work to interpret the results.

Obfuscation, Packing and Polymorphism. As shown in [28], and as motivated by the subsequent research in the field, the main drawback of static analysis arise when the malware authors transform their code [41], which is a longstanding effective practice to fool static signatures employed by current detectors deployed on the market.

Several approaches have been proposed to counteract obfuscation [23] and polymorphism. An example relevant to our work is [19], which shows that the connected sub-graph of a given size of the CFG are robust against polymorphism. We use a variant of this technique in our own work.

Binary Analysis A significant example of hybrid analysis is Reanimator [6], which finds implementations of a dynamically observed behavior in samples that did not exhibit it, to unveil the so-called "dormant" functionalities. Relevant behaviors are detected by means of manually-written specifications. Reanimator creates a model of the identified code regions using the same CFG fingerprints we use [19]. With these hybrid models Reanimator statically checks whether another (unpacked) binary contains similar code. The key difference of our system is that we extract (relevant) high-level behavior graphs automatically, without needing manually-written specifications.

Another relevant work is Beagle [21], where the authors observe that malware authors regularly update their software in order to beat defenses, improve their capabilities or change their business model. The goal of Beagle is to observe the evolution of a malware family over time. For this, it regularly downloads new versions of the same malware, then compares them with the older ones through a series of static and dynamic diffing techniques. Then, Beagle maps found differences back to the implementation code for further analysis. Beagle also assumes that a set of high-level behaviors are available, which the authors must define manually.

We cited earlier [22]. Central to this work is once again the concept of behavior: The authors collect multiple execution traces on different environments and deem a malware as evasive if such traces differ. From this work, we can conclude that inferring *differences* between unknown behaviors automatically is feasible, whereas detecting *behaviors* automatically is still unexplored.

Analysis and Detection. An interesting approach that exploits dynamic analysis for malware detection is Panorama [42]. It executes unknown programs in a out-of-the-box installation of Microsoft Windows, using scripts to introduce sensitive information in the system. Then, it tracks propagation of this information, thanks to dynamic analysis, summarizing it in so called taint graphs. Panorama discerns malicious from benign behavior by manually defined detection policies. This and similar works are complementary to JACKDAW, since our goal is defining interesting behaviors automatically.

Behavior Extraction. The work described in [11] has a similar goal to ours. However, they concentrate on the dynamic aspect of behaviors: our behavior specifications are richer and more contextualized, as they include static information obtained by correlating many variant implementations of the same behavior. Moreover, the static information allows static matching, which is faster and, more importantly, accounts for dormant code. Another difference is that they focus on malicious binaries, whereas we do not make any apriori assumption on the intent of a program. From a technical viewpoint, their approach is to build a dependency graph from dynamic analysis of malicious and benign binaries. Then, they exploit structural leap mining to discern between malicious and benign behaviors using dependency graph built running benign code. Finally, concept analysis is used to synthesize the rule that will represents behaviors that are in malicious software e not in benign ones.

A recent related work [32] proposes to extract behaviors as significant sub-graphs of the system call dependency graph. The key intuition is that graphs of goodware and malware will exhibit substantially different features, which can be used train a classifier able to extract both known and unseen behaviors from new binaries. Although this approach is related to ours, we leverage the arguments and the return values of the API calls to find interesting sub-graphs and provide a semantic meaning of the behavior extracted.

7 Conclusions

JACKDAW is able to automatically find groups of API calls that represent high-level actions, which we call behaviors, exploiting hybrid analysis on a large dataset of binaries. It maps such behaviors on the code, using fingerprints suitable for subsequent static analysis and resilient to basic code transformations. We leverage web knowledge bases to annotate behaviors with a series of hints about their nature, by means of semantic tags that support analysts in understanding what they are seeing.

We showed auto-consistency between static and dynamic components of such behaviors. Also, we showed that automatically-generated behaviors could be matched against a ground truth defined with the helps of expert analysts, resulting in 34 out of 44 manually defined behaviors being matched and automatically discovered by JACKDAW. Then, we verified by manual inspection and by means of a panel of experts that extracted behaviors are meaningful, and that their semantic tags are consistent.

As a future research direction, the outputs of our system can be used as an input to other hybrid analysis systems, or to augment reverse-engineering tools in order to automatically annotate the portions of the CFG that implement behaviors.

References

1. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. *NDSS*, 2009.
2. U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *LEET*, 2009.
3. D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. Hi-cfg: Construction by binary analysis and application to attack polymorphism. In *ESORICS*, pages 164–181, 2013.
4. S. Cesare and Y. Xiang. *Software Similarity and Classification.* Springer Briefs in Computer Science. Springer, 2012.
5. S. Cesare, Y. Xiang, and W. Zhou. Control flow-based malware variant detection. *Dependable and Secure Computing, IEEE Transactions on*, PP(99):1–1, 2013.
6. P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *SP*, pages 61–76, Washington, DC, USA, 2010. IEEE Computer Society.
7. J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *TACO*, 3(4):359–389, Dec. 2006.
8. Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *ACSAC*, New York, NY, USA, December 2013.
9. B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *SP*, pages 297–312, 2011.
10. M. Eskandari, Z. Khorshidpour, and S. Hashemi. Hdm-analyser: a hybrid analysis approach based on data mining techniques for malware detection. *JCV*, 9(2):77–93, 2013.
11. M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *SP*, pages 45–60, Washington, DC, USA, 2010. IEEE Computer Society.
12. Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *SP*, pages 586–600, 2012.
13. T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *HOTOS*, pages 6:1–6:6, Berkeley, CA, USA, 2007. USENIX Association.
14. T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *6th IEEE SMC Information Assurance Workshop*, 2005.
15. G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *DIMVA*, pages 102–122, Berlin, Heidelberg, 2013. Springer-Verlag.
16. G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *JCV*, 4(3):251–266, 2008.
17. J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *USENIX Security*, pages 81–96, Berkeley, CA, USA, 2013. USENIX Association.
18. D. Kirat, G. Vigna, and C. Kruegel. Barebox: efficient malware analysis on bare-metal. In *ACSAC*, pages 403–412, New York, NY, USA, 2011. ACM.
19. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, pages 207–226, Berlin, Heidelberg, 2006. Springer-Verlag.
20. J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*, 2011.
21. M. Lindorfer, A. D. Federico, F. Maggi, P. M. Comparetti, and S. Zanero. Lines of malicious code: Insights into the malicious software industry. In *ACSAC*, pages 349–358, New York, NY, USA, December 2012. ACM.

22. M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti. Detecting environment-sensitive malware. In *RAID*, pages 338–357, Berlin, Heidelberg, 2011. Springer-Verlag.
23. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, pages 290–299, New York, NY, USA, 2003. ACM.
24. F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *TODS*, 7(4):381–395, 2008.
25. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *ACSAC*, pages 431–441. IEEE, 2007.
26. L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID*, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.
27. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *SP*, 2007.
28. A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *ACSAC*, pages 421–430, 2007.
29. D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *TISSEC*, 9(1):61–93, Feb. 2006.
30. K. Nance, M. Bishop, and B. Hay. Virtual machine introspection: Observation or interference? *Security Privacy, IEEE*, 6(5):32–37, 2008.
31. J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*. Internet Society, 2005.
32. S. Palahan, D. Babic, S. Chaudhuri, and D. Kifer. Extraction of statistically signicant malware behaviors. In *ACSAC*, New York, NY, USA, December 2013.
33. K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *JCS*, 19(4):639–668, Dec. 2011.
34. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.
35. E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security*, 2013.
36. A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*. Citeseer, 2011.
37. 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 *ICISS*, Hyderabad, India, Dec. 2008.
38. Q. Song and N. Kasabov. Ecm - a novel on-line, evolving clustering method and its applications. In *In M. I. Posner (Ed.), Foundations of cognitive science*, pages 631–682. The MIT Press, 2001.
39. C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan. Down to the bare metal: using processor features for binary analysis. In *ACSAC*, pages 189–198, New York, NY, USA, 2012. ACM.
40. G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *DIMVA*, pages 41–61, Berlin, Heidelberg, 2013. Springer-Verlag.
41. T. Yetiser. Polymorphic Viruses, Implementation, Detection, and Protection, 1993.
42. H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *CCS*, pages 116–127. ACM, 2007.