mantically similar and related flows between system entities during the construction of our data flow graphs, we can furthermore keep the underlying data structures light-weight and efficient. Considering the findings of Frederikson et al. [11] that malware detection through behavioral matching in general cannot be done efficiently, simplifications as used in our approach are particularly helpful in keeping detection efficiency within reasonable boundaries.

**Problem:** In sum, the main problem tackled by this paper is to efficiently and effectively detect malware and malicious activities while being resilient against behavioral obfuscation techniques as used by current malware.

**Solution:** We tackle this problem with a generic behavioral malware detection approach that is based on the analysis of quantitative data flows at the OS level. By aggregating and abstracting concrete system events into resulting data flows between relevant system entities, we build graphs that represent the data flow behavior of all entities of a system. Based on these graphs we then specify and later on try to detect data flow patterns that are characteristic for specific types of malware. As a proof-of-concept we show an instantiation of our generic approach for Microsoft Windows operating systems.

**Contributions:** The original contributions of this paper are thus as follows: **a)** To the best of our knowledge, we are the first to make use of quantitative data flow analysis for detecting malware. **b)** We introduce a generic model to specify malware detection heuristics based on aggregated data flow graphs. **c)** We present a proof of concept instantiation of this model to detect malware in Microsoft Windows operating systems and **d)** We show that our approach is efficient (because of comparably light-weight data structure due to underlying abstractions and aggregations) and effective (a proof-of-concept evaluation indicates a high detection rate and precision).

**Organization.** In Section 2 we introduce the general ideas and present the data flow model and graph construction as well as the corresponding malware detection concept. In Section 3 we instantiate our approach for Microsoft Windows operating systems and present representative data-flow based malware detection heuristics. In Section 4 we analyze the runtime and detection performance of our prototype. In Section 5 we then give an overview of related work in the area of behavioral malware detection and put our work into context. Finally we conclude in Section 6.

# 2. PROPOSED APPROACH

In this section we present our aggregated quantitative data flow approach for malware detection and analysis. We proceed in two steps. We start by presenting a generic model that can be instantiated for various system types, and that provides a data structure suitable for data flows analysis. Then, we discuss how to use this data structure for the task of malware detection.

The basic idea is to base malware detection on the analysis of system-wide quantitative data flows. System-wide data flows in this context mean all flows of data between different system entities that happened within a specific time frame. *System entities* denote here all conceptual sources and sinks of data. In the case of operating systems, this includes resources like processes, sockets, or files. Data flows between system entities are caused by the execution of specific data flow related events. For operating systems, an example is file system events that, if called by a process, lead to a flow of data between the calling process and the involved file.

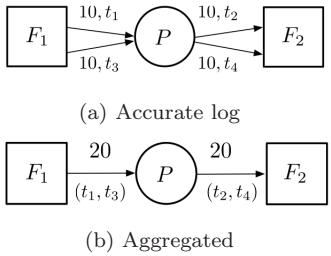



Figure 1: Abstraction from single system events

We describe such system-wide data flows with data flow graphs. Our data flow graphs contain nodes and edges: nodes represent system resources that were at least once involved in a data flow; and edges represent the actual data flows between these entities.

*Example* Suppose process $P$ reads 10 Bytes from file $F_1$, then writes them to file $F_2$, and it repeats this process twice. An accurate log of this events would keep track of each single system event and their timestamp, as depicted in Figure 1a. As a built-in optimization, we aggregate these flows into single weighted events as depicted in Figure 1b together with a record of the first system event and the last one.

In Section 2.1 we formalize this notion and the graph updating algorithm. In Section 2.2 we discuss how to analyze such graphs with graph invariants.

## 2.1 Generic Data Flow Model

Our goal is to perform malware detection with quantitative data flow graphs (QDFGs), denoted by $\mathcal{G}$. They represent data flows (edges) in-between relevant resources of a system (nodes). QDFGs are incrementally built by runtime monitors that capture all relevant system-level events, $\mathcal{E}$, that move data from one resource to another, and therefore induce a change of the QDFG. QDFGs hence evolve over time: events cause either the creation or the update of nodes or edges in a data flow graph.

More precisely, QDFGs are elements of the set $\mathcal{G} = \overline{N} \times \overline{E} \times \overline{A} \times ((\overline{N} \cup \overline{E}) \times \overline{A} \to \mathit{Value}^{\overline{A}})$ for a set of nodes, $\overline{N}$, a set of edges, $\overline{E} \subseteq \overline{N} \times \overline{N}$, a set of attribute names, $\overline{A}$, and a set of labeling functions drawn from $((\overline{N} \cup \overline{E}) \times \overline{A}) \to \mathit{Value}^{\overline{A}}$ that map an attribute $a \in \overline{A}$ of a node or an edge to a value drawn from set $\mathit{Value}^{\overline{A}}$.

In a QDFG $G = (N, E, A, \lambda) \in \mathcal{G}$, nodes $N$ represent data flow related system entities and edges $E$ data flows between them. Attributes $A$ are needed to keep our model flexible enough to be instantiated for various types of systems. They represent characteristics of data flows and system entities that are important for malware detection and analysis. Edges, for instance, have an attribute *size* that represents the amount of transferred data of the respective flow. The labeling function $\lambda$ retrieves the value of an attribute assigned to a node or an edge, in this example the size of the flow that corresponds to an edge.

QDFGs are intuitively to be read as follows: If there has been a flow of data in-between the system entities corresponding to two nodes, there is an edge between these nodes. Data flows are caused by system events. Among other things, this is the case if a process reads from a file; writes to a registry; or writes to a socket. At the level of

a QDFG, we are not interested precisely which event has caused the flow. Instead, we model events $(src, dst, size, t, \lambda) \in \mathcal{E}$ as tuples where $src \in \overline{N}$ is the originating system entity, $dst \in \overline{N}$ the destination system entity, $size \in \mathbb{N}$ is the amount of transferred data, $t \in Value^{time}$ is a time-stamp (defined below) and $\lambda$ holds other attributes of the involved entities. Examples will be provided in Section 3.1.2. As mentioned above, a runtime monitor will then observe all events at, say, the OS level, and use the characterization of an event by set $\mathcal{E}$ as a data transfer to update a QDFG.

In order to define how exactly the execution of an event modifies a QDFG, we need some notation for updating attribute assignments. For any node/edge, attribute pair $(x, a) \in (N \cup E) \times \overline{A}$, we define $\lambda[(x, a) \leftarrow v] = \lambda'$ with $\lambda'$ identical to $\lambda$ for all values but for $(x, a)$ where $\lambda'(x, a) = v$.

For brevity's sake, we agree on some syntactic sugar to represent multiple synchronous updates:

$$\lambda[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_k] = (\ldots(\lambda[(x_1, a_1) \leftarrow v_1])\ldots)[(x_k, a_k) \leftarrow v_n].$$

The composition of two labeling functions is defined as:

$$\lambda_1 \circ \lambda_2 = \lambda_1[(x_1, a_1) \leftarrow v_1; \ldots; (x_k, a_k) \leftarrow v_n]$$

where $v_i = \lambda_2(x_i, a_i)$ and $(x_i, a_i) \in \mathsf{dom}(\lambda_2)$.

We consider the aggregation of flows and system entities to be one distinct feature of our approach. Such aggregations are needed to keep the resulting graphs within reasonable limits as illustrated in Fig. 1. This in particular means that entities with the same name (e.g. multiple running instances of the same program) are represented by the same node. Furthermore, flows between the same pair of system entities are represented by one edge where we simply sum the resulting transferred amount of data rather than creating two different edges.

In order to aggregate flows caused by similar events between the same entities edges get assigned a time interval attribute $time \in \overline{A}$ such that $Value^{time} \subseteq \mathbb{N} \times \mathbb{N}$. This interval represents the first and the last point in time where an event was executed that either led to the creation or update of one specific edge as illustrated in Fig. 1b.

Using the auxiliary functions $min, max : Value^{time} \to \mathbb{N}$ with $min((t_1, t_2)) = t_1$ and $max((t_1, t_2)) = t_2$ we can finally precisely define how an event updates a QDFG by function $update : \mathcal{G} \times \mathcal{E} \to \mathcal{G}$ defined in Figure 2.

$$update(G, (src, dst, s, t, \lambda')) = \begin{cases} \begin{pmatrix} N, \\ E, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow \lambda(e, size) + s; \\ (e, time) \leftarrow (min(\lambda(e, time)), t) \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{if} \quad e \in E \\ \begin{pmatrix} N \cup \{src, dst\}, \\ E \cup \{e\}, \\ A \cup \mathsf{dom}(\lambda'), \\ \lambda \begin{bmatrix} (e, size) \leftarrow s; \\ (e, time) \leftarrow (t, t) \end{bmatrix} \circ \lambda' \end{pmatrix} & \text{otherwise} \end{cases}$$

where $e = (src, dst)$ and $G = (N, E, A, \lambda)$

Figure 2: Graph update function

In addition to the state update function we need a few auxiliary functions as basis for the definition of malware detection heuristics. Because their definition is standard, we omit a formalization. Function $pre : \overline{N} \times \mathcal{G} \to 2^{\overline{N}}$ computes all immediate predecessor nodes of a node of the graph. Conversely, $suc : \overline{N} \times \mathcal{G} \to 2^{\overline{N}}$ computes the immediate successors of a node. Functions $in, out : \overline{N} \times \mathcal{G} \to 2^{\overline{E}}$ compute the set of incoming and outgoing edges of a node. If $P$ denotes the set of all finite sequences, function $paths_{from} : \overline{N} \times \mathcal{G} \to 2^P$ calculates all (temporally consistent) paths without loops that originate from a given node $n$. The intuition is that there is a data flow from $n$ to each node in the path. Conversely, and finally, function $paths_{to} : \overline{N} \times \mathcal{G} \to 2^P$ computes all those paths without loops in the graph that have the argument node as last element. For the accuracy of our analysis it is important that paths satisfy a temporal consistency property (for a discussion on soundness of this approximation see Appendix A).

## 2.2 Malware Detection

We now have a basis for specifying malware-specific (quantitative) data flow heuristics. These heuristics will enable us to discriminate between benign and potentially malicious system entities. Typical examples for such heuristics include restrictions on orders or specific sequences of flows that are characteristic for certain malware types.

We express such malware data flow heuristics as first-order logic predicates on properties of the introduced QDFGs. The set $\Phi_{\text{mw}} := \Phi_{\text{rep}} \cup \Phi_{\text{mpl}} \cup \Phi_{\text{qnt}}$ contains all specified malware replication, manipulation, and quantitative heuristics that model specific classes of malware behavior. We will define these heuristics in Section 3.2 and formalize them in Appendix B. The motivation for our separation of heuristics is that we want to differentiate between heuristics that detect replication and those that detect the system manipulation behavior of a malware. This separation also allows us to increase the detection specificity by combining different types of heuristics. In addition we will also introduce quantitative heuristics to leverage quantitative data flow aspects in order to increase detection rates. Note that the sets of heuristics are not necessarily disjoint.

The detection of potentially malicious entities at runtime is then done by (1) continuously updating the QDFG and (2) by iterating over all system entities in the graph and matching the specified detection heuristics against them.

More specifically, for an entity to be considered malicious at least one replication and either a manipulation or quantitative heuristic must match.

To use our generic approach for platform–specific detection of malware one must provide platform–specific instantiations of both, data flow model and detection heuristics. This in particular means that (1) all interesting entity types of the target platform must be mapped to nodes in our model; and that (2) all system events must be mapped to corresponding changes of QDFGs as defined by the update function. Furthermore platform–specific instances of abstract heuristics must be specified, based on the previously defined system entities and events.

# 3. INSTANTIATION FOR WINDOWS

To demonstrate feasibility and effectiveness of our approach, we instantiated and implemented our generic model

for the Microsoft Windows operating system family. We identified relevant system entities as potential sources or sinks of malware activities as well as system events that lead to data flows between them. Based on these entity and event definitions, we specify several Windows-specific data flow based malware detection heuristics.

To gather the necessary runtime information, we made use of a user-space monitor based on Windows API function call interposition. It intercepts all process interactions with respect to the identified set of relevant system events. More details on the design and technical implementation of the monitor can be found in [31].

## 3.1 Instantiated Model

### 3.1.1 System Entities

Based on a inductive study on process interactions and resulting data flows within Windows operating systems we identified a set of system resources that can be considered as sources or sinks of data flows, and thus as entities in our generic model. We use an attribute *type* $\in \overline{A}$ to describe an entity's type: **Processes** interact with the file system, registry, and sockets and cause data flows from or to these entities. Nodes that represent processes are assigned $P$ as value of the *type* attribute. **Files** persistently store data and can be either read or written to by processes. Nodes that represent files are always implicitly assigned $F$ as value of the *type* attribute. **Sockets** are connection points to remote systems. Processes can either read or write data from them, causing data flows from or to the respective remote systems. Socket nodes are always assigned $S$ as value of the *type* attribute. **Registry Keys** persistently store settings and other Windows configuration data and can be either read or written to by processes. Nodes that represent registry keys are assigned $R$ as value of the *type* attribute.

### 3.1.2 System Events

To model interactions that are relevant from a data flow perspective, we model several Windows API functions by abstract events. These will be used in the update function of Section 2.1, and thus by the runtime monitor, to build the QDFG. Due to the considerable complexity of the Windows API we concentrate on a subset of the Windows API that is commonly used by malware to interact with system resources. For brevity's sake we sometimes simplify some aspects of these functions like parameter types (e.g. use file names instead of file handlers) or enriched them with additional information that is usually not directly given by the function's parameters.

In the following we only describe one representative of each type of events which model all semantically (in terms of induced data flows) equivalent events; e.g. the WriteFile function that represents all Windows API functions that induce a data flow from a process to a file. In the prototypical implementation of our approach we considered and intercepted a wide range of semantically equivalent events for each class.

**File System Operations**

- **ReadFile(Ex)** Using this function a process reads a specified amount of bytes from a file to its memory.
  **Relevant Parameters**: Calling Process ($P_C$), Source File ($F_S$), ToReadBytes ($S_R$), File Size ($S_F$)
  **Mapping**: $(F_S, P_C, S_R, t, \emptyset[(F_S, size) \leftarrow S_F]) \in \mathcal{E}$
- **WriteFile(Ex)** Using this function a process can write a specific number of bytes to a file.
  **Relevant Parameters**: Calling Process ($P_C$), Destination File ($F_D$), ToWriteBytes ($S_W$), File Size ($S_F$)
  **Mapping**: $(P_C, F_S, S_W, t, \emptyset[(F_D, size) \leftarrow S_F]) \in \mathcal{E}$

**Registry Operations**

- **RegQueryValue(Ex)** Using this function a process reads the value of a specific registry key.
  **Relevant Parameters**: Calling Process ($P_C$), Source Key ($K_S$), ToReadBytes ($S_R$)
  **Mapping**: $(K_S, P_C, S_R, t, \emptyset) \in \mathcal{E}$
- **RegSetValue(Ex)** Using this function a process can write data to a specific registry key.
  **Relevant Parameters**: Calling Process ($P_C$), Destination Key ($K_D$), ToWriteBytes ($S_W$)
  **Mapping**: $(P_C, K_S, S_W, t, \emptyset) \in \mathcal{E}$

**Socket Operations**

- **Recv** Using this function a process can read a specific number of bytes from a network socket.
  **Relevant Parameters**: Calling Process ($P_C$), Source Address (IP Port) ($A_S$), ToReadBytes ($S_R$)
  **Mapping**: $(A_S, P_C, S_R, t, \emptyset) \in \mathcal{E}$
- **Send** Using this function a process can send a specific number of bytes to a network socket.
  **Relevant Parameters**: Calling Process ($P_C$), Destination Address (IP Port) ($A_D$), ToWriteBytes ($S_W$)
  **Mapping**: $(P_C, A_D, S_W, t, \emptyset) \in \mathcal{E}$

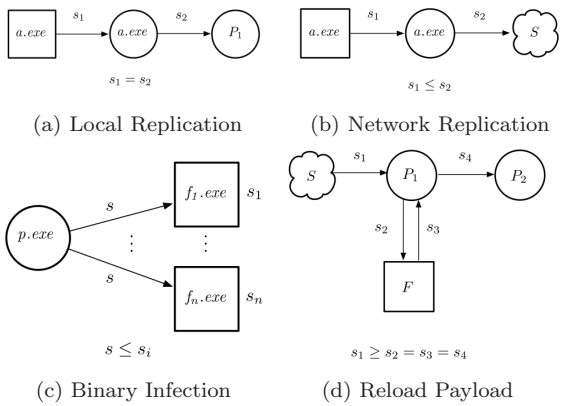
**Process Operations**

- **CreateProcess(Ex)** Through this function a process can trigger the creation of another process, using a specific executable file as binary image.
  **Relevant Parameters**: Caller Process ($P_C$), Callee Process ($P_D$), Binary Name ($F_B$), Binary Size ($S_B$)
  **Mapping**: $(P_C, P_D, S_B, t, \emptyset[(P_D, size) \leftarrow S_B]) \in \mathcal{E}$
- **ReadProcessMemory** Using this function a process can read a specific number of bytes from the memory of another process.
  **Relevant Parameters**: Calling Process ($P_C$), Source Process ($P_S$), ToReadBytes ($S_R$)
  **Mapping**: $(P_D, P_C, S_R, t, \emptyset) \in \mathcal{E}$
- **WriteProcessMemory** By this function one process can write a specific number of bytes to the memory of another one.
  **Relevant Parameters**: Calling Process ($P_C$), Destination Process ($P_D$), ToWriteBytes ($S_W$)
  **Mapping**: $(P_C, P_D, S_W, t, \emptyset) \in \mathcal{E}$

## 3.2 Heuristics

Based on the generic data flow model, Windows system entities and the mapping of Windows API calls to events, we can now present specific malware detection heuristics that make use of (quantitative) data flows and data flow properties. We used these for our prototype to detect potentially malicious processes and for the experiments in Section 4. These heuristics were deductively defined on the basis of malware behavior databases [28] and academic malware analysis reports [2]. In the following, we will say that a "heuristic triggers" if according to this heuristic, malware is present.

The basic idea of constructing such heuristics is to identify characteristic data flows and flow properties that correspond to typical high-level behavior of malware such as "*a malicious process tries to replicate itself by infecting other benign binaries*", like e.g. seen for the Parite worm, or "*a malicious process tries to replicate itself via email*", as e.g.




(a) Local Replication (b) Network Replication

(c) Binary Infection (d) Reload Payload

Figure 3: Replication Heuristics

common for email worms like MyDoom. These characteristic data flows or properties than represent a set of potentially malicious activities that can later be re-identified in data flow graphs generated at runtime.

Not all of these heuristics in themselves are sufficiently specific and expressive to always discriminate between malware and goodware. An example for this is a heuristic that triggers if a process downloads payload from the internet and starts it as a new process. Although such behavior is typical for so-called dropper malware that install additional malware (so-called eggs) using this technique, it also matches the behavior of benign web installers. Yet, by combining several heuristics, we can achieve a sufficient specificity and precision do correctly discriminate between good- and malware in most cases.

As we focus on the detection of Windows malware in this paper, the following heuristics are specifically designed for Windows operating systems and make use of Windows-specific flow properties. Nevertheless many of these heuristics can be generalized to other operating systems as they e.g. resemble generic high-level worm replication behavior that is also typical for Linux-based malware.

Each of the following heuristic specification consists of a natural language description of the corresponding flow pattern, and a brief description of the rationales behind the heuristic. In addition we visually illustrated some basic flow patterns that correspond to these heuristics (see Figure 3), where boxes represent file nodes, circles represent process nodes, and clouds represent socket nodes. Due to space limitations we restricted the visual descriptions to the category of replication heuristics.

Formalizations of all heuristics are provided in Appendix B.

### 3.2.1 Replication Heuristics

The heuristics in this class capture activities targeting infection of a local or remote system with malware. That can e.g. be achieved by appending malicious binaries to a set of benign system programs, injecting malicious code into other processes, or sending binaries to remote network nodes.

**Local Replication ($\phi_{\text{rep}_1}$)** Triggers if a process node has at least one flow from a file node that has the same name as the process, followed by at least one flow to another process. To increase the precision of the pattern the amount of flown data from file to process must be the same as the amount of data that flows between the two processes (as e.g. observed for Trojans like Agent). *Rationale:* This heuristic covers characteristic data flows caused by a malicious process trying to replicate itself by spawning clone processes from the own binary image.

**Network Replication ($\phi_{\text{rep}_2}$)** Similar to the local replication pattern, this heuristic triggers if there exists at least one flow between a process and a file of similar name and a flow from this process to a network socket. To increase specificity the amount of data sent to the socket must be at least as big as the amount of data read from the file (as e.g. observed for Email Worms like MyDoom). *Rationale:* This heuristic covers data flows that are typical for a malware that tries to replicate itself by sending its binary image over the network to infect remote systems.

**Binary Infection ($\phi_{\text{rep}_3}$)** This heuristic triggers if there exist multiple flows (at least two) from a process to executable binary files. To reduce false positives an additional quantitative constraint is as follows. The amount of transferred data to the benign executables must be at least as high as the amount of data from the binary image of the process to the process; and the size of the target binary files must be greater than 0. This to some extent ensures that at least the size of the malware image is appended to already existing binaries. *Rationale:* These data flows resemble malware activities that are targeted at replication through infection of other program's executables. This usually happens through malware appending its own code to benign binaries (as e.g. observed for Viruses like Parite).

**Download and Start Payload ($\phi_{\text{rep}_4}$)** This heuristic trigers if there is a flow from a socket node to a process node, a flow from this process to a executable binary file node, a flow from this file node back to the process node, and then a flow from this process node to a new process node with a similar name as the file node. To increase detection specificity we increase the additional quantitative constraint that all flows of this pattern must have the same quantities, except for the first flow from the socket node that may also be bigger due to additional meta- and network control data. This to some extent ensures that at least the downloaded size of payload is propagated to a binary file and then to a new malicious process. *Rationale:* This data flow pattern subsumes malware behavior that targets reloading and executing additional malicious payload from the internet (observed for Droppers as e.g. used by the Conficker virus).

### 3.2.2 Manipulation Heuristics

This class of heuristics contains certain flow patterns that correlate with specific high-level semantics for activities that fall under the broad category of manipulation of system integrity or data confidentiality. Such heuristics for example include activities that target leaking sensitive data to untrusted locations like the internet, modifications of the registry to e.g. add autostart entries, or opening backdoors for further malicious activities.

**Leak Cached Internet Data ($\phi_{\text{mpl}_1}$)** Whenever we detect a flow of data from a dedicated internet cache file (in our prototype identified by the absolute file path containing either "*Cookies*" or "*Temporary Internet Files*" as substring) to a process that afterwards sends data to a socket, this heuristic triggers. The specificity of this heuristic is increased by demanding that the flow from the leaking process to the remote socket node must be as least as big as the flow from the cache file to the process. *Rationale:* This heuristic captures the data flow behavior of a malicious process trying

to steal potentially sensitive data like cookies from dedicated internet cache folders by sending them to a remote network location (as e.g. observed for various samples of the generic Infostealer malware family).

**Spawn Shell ($\phi_{\text{mpl}_2}$)** This heuristic triggers if a command line shell process node (in Windows identified by the name of the process node containing the sub-string "*cmd.exe*") has an incoming or outgoing data flow connection to at least one socket. For simplicity's sake we currently only consider processes with an indirect connection with a maximum distance of at maximum two hops to a socket. *Rationale:* This heuristic describes data flows caused by malware trying to glue a command shell to a listening socket to open a backdoor (as e.g. observed for Backdoors like Autorun).

**Deploy System Driver ($\phi_{\text{mpl}_3}$)** This heuristic triggers if we detect a flow between a process and a system driver binary, identified by its name containing the system driver extension "*.sys*", followed by a flow from this system driver file to one of Window's driver loading utilities (in our current implementation we consider "*wdreg.exe*" and "*sc.exe*"). To increase the specificity of this heuristic we also demand that the flow from the potentially malicious processes to the system driver file and from the file to the driver loading utility node must be of the same size. *Rationale:* Today, many sophisticated malware types make use of root kit technology to take over full control of a compromised system and hide their behavior from anti malware software. This heuristic thus describes the data flows that correlate with malware attempts to deploy and load malicious system drivers to the Windows kernel to inject its root kit functionality (as e.g. seen for the ZeroAccess root kit).

### 3.2.3 Quantitative Heuristics

The heuristics of this class capture typical quantitative data flow properties of malware to discriminate malicious processes from benign ones. Examples for such properties are characteristics of distributions or quantitative relationships between edge or node types. The thresholds for these heuristics were inductively derived through analyzing a set of eleven malware samples, distinct from the ones used for the evaluation.

**Single-hop Data Amount Entropy ($\phi_{\text{qnt}_1}$)** This heuristic calculates the normalized entropy of the data amount distributions of all outgoing edges of a potentially malicious process and triggers if the corresponding distribution is too uniform. More precisely, for this heuristic to trigger, the normalized entropy (actual entropy divided by hypothetical entropy under uniform distribution) of the measured distribution must be higher than 0.8, indicating an almost uniform outgoing data flow distribution. *Rationale:* A lot of malware types replicate themselves by infecting other binaries or injecting code into other system processes. This often results in almost uniform data amount distributions of the outgoing edges of the corresponding graph nodes.

**Registry Call Ratio ($\phi_{\text{qnt}_2}$)** This quantitative heuristic calculates the relative ratio of flows from the Windows registry with respect to the overall communication of a process. A high percentage of registry-related edges within all in-edges of a node is considered indicator for malicious activities. A threshold for this that turned out to work well in practice was a registry edge ratio of 0.8. *Rationale:* Malware often performs a lot of registry queries to probe the state of the infected system before starting its actual malicious activities. This results in a comparably high ratio of registry-related flows in the first moments after process startup.

| Malware Family | All | Act. | Det. (N/Q) | Det.Rt.(all) (N/Q) | Det.Rt.(act.) (N/Q) |
|---|---|---|---|---|---|
| Agent | 2 | 2 | 2/2 | 100% / 100% | 100% / 100% |
| Agobot | 6 | 4 | 4/4 | 67% / 67% | 100% / 100% |
| Autorun | 7 | 4 | 4/4 | 57% / 57% | 100% / 100% |
| Bagle | 5 | 1 | 0/1 | 0% / 20% | 0% / 100% |
| Bancos | 5 | 2 | 2/2 | 40% / 40% | 100% / 100% |
| Infostealer | 2 | 2 | 2/2 | 100% / 100% | 100% / 100% |
| MyDoom | 4 | 2 | 0/1 | 0% / 25% | 0% / 50% |
| Parite | 7 | 3 | 2/3 | 29% / 43% | 67% / 100% |
| SpyEye | 6 | 3 | 3/3 | 50% / 50% | 100% / 100% |
| **Total** | 44 | 23 | 19/22 | 43.18% / 50% | 82.61% / 95.65% |

Table 1: Detection Rate

## 4. EVALUATION

In general, it is difficult to objectively assess the effectiveness and precision of any behavior-based malware detection technique. This is because experiment outcomes heavily depend on chosen goodware and malware evaluation sets, malware activity during behavior recording, and counter-analysis measures employed by the malware.

To shed light on the general feasibility of our concepts and to evaluate our prototype we thus conducted a proof-of-concept study based on a representative set of goodware and malware. Within this study we investigated the effectiveness and efficiency of our detection approach. In terms of effectiveness we mainly focused on detection and false positive rates, where detection rate is defined as the fraction of correctly detected malware samples in all analyzed malware samples, and false positive rate by the ratio of goodware samples, wrongly classified as malware, within the entire set of analyzed goodware samples. In terms of efficiency we analyzed average detection time and space consumption.

We conducted two types of tests. In both we installed malware or goodware samples within our monitored environment and recorded their behavior in terms of calls to the Windows API to dedicated log files. These log files then built the basis for our actual graph generation and heuristic matching approach that in the end lead to the classification of processes into malicious and benign processes.

**Malware Tests:** For this type of tests we manually installed various samples from a representative set of malware families and recorded their behavior. In addition, we launched a small set of benign applications to increase the interaction diversity, introduce some non-determinism of the overall system behavior, and thus limit the detection bias for the further processing steps. The main goal of these test was to get a sufficiently large data base of malicious processes to reason about detection precision, in particular to derive an average false negative rate. To analyze a representative set of malware types, we picked samples from a public malware database[1] according to the prominent malware types as mentioned on the website of an anti-malware vendor[2].

[1] `http://openmalware.org/`

[2] `http://www.symantec.com/security_response/`

**Goodware Tests:** The intention of this type of tests was to get a data basis for reasoning about false positives. For this purpose we installed and executed a set of knowingly benign programs into our monitored environment. To increase the diversity of the behavior logs we furthermore performed simple tasks like opening or saving files using these programs. To get a representative set of analyzed goodwares, we installed the 20 most popular programs from `http://download.com`. In addition, we used multiple standard Windows programs that were already pre-installed on the evaluation machine.

All tests were conducted on a Windows 7 Service Pack 1 installation, running within a Oracle VirtualBox instance that was assigned two 2.7 Ghz cores and 4GByte of RAM. The host system itself was running a Windows 8 installation powered by a Intel i7 Quadcore 2.7 Ghz CPU and 8 GByte of RAM. The data collection part was done within the Windows 7 VirtualBox instance, whereas the actual behavioral analysis was conducted offline on the host system.

Note that our detection approach does not require an initial learning phase as it entirely depends on deductively defined set of static data flow heuristics. As basis for the subsequent analysis we performed 44 malware and 21 goodware tests, resulting in 65 activity logs, each containing a trace of process behavior monitored over a period of 5 minutes.

## 4.1 Effectiveness

For each of the collected malware and goodware logs we generated a quantitative data flow graph, as described in Section 2.1. The corresponding malware test graphs thus represented the behavior of 44 unique samples of the 10 malware families: Agent, Agobot, Autorun, Bagle, Bancos, Infostealer, MyDoom, Parite, and SpyEye.

One of the reasons for the comparably limited evaluation set of 10 different malware families is in the fact that we conducted our tests in a realistic state-of-the-art execution environment and thus picked Windows 7 SP1 as evaluation platform. This choice made it particularly hard to install many malware types like Nimda, Ramnit, or NetSky, because of their incompatibility with the used OS.

Moreover, some of the analyzed malware types had built-in anti-analysis measures that for example detected the presence of certain virtual machine files and settings and due to that stopped all further activities [24]. One example for this were the various samples of the Sality worm that we failed to analyze due to their awareness of being executed in a VirtualBox environment.

Furthermore, many of the analyzed malware samples did not show any significant activities during the 5 minute monitoring period. This inactivity was due to event-based activity mechanisms that were not triggered during our analysis; the corresponding malware was aware of being analyzed and thus did not start its usual infection routines; or it had deferred activity that we simply did not catch within our short monitoring period. This led us to further subdivide the set of malware samples into active and inactive ones. As our approach entirely relies on analysis of runtime behavior, we do not consider non-detected inactive malware samples to be real false negatives.

For the goodware tests we constructed graphs that represented the behavior of 2199 samples of 131 different known benign programs, ranging from standard pre-installed Windows programs like notepad or paint, over various installation routines for software like Oracle's Java, or Adobe's Flash Player, to a variety of third party software, including Microsoft Office products, anti-virus scanners, webcam software, or archiving or music production tools.

| Goodware Family | Smp. | FP (N/Q) | FP Rate (N/Q) |
|---|---|---|---|
| Installer | 261 | 14/18 | 5.36% / 6.90% |
| Std. Programs | 1194 | 2/6 | 0.17% / 0.50% |
| 3rd Party | 744 | 6/11 | 0.81% / 1.48% |
| **Total** | 2199 | 22/35 | 1.00% / 1.59% |

Table 2: False Positive Rate

On these graphs we then conducted the analysis steps, as described in Section 2.2, using the heuristics described in Section 3.2, to classify each graph node into the categories *malicious* and *benign*. To further analyze the effect of quantitative properties on detection effectiveness we conducted the test both with and without using quantitative heuristics. As depicted in Table 1, we were able to detect all analyzed malware families. In sum, with respect to the active malware samples that we analyzed, we achieved a detection rate of about 83% without, and about 96% with using quantitative heuristics. This clearly underlines our hypothesis on the usefulness of quantitative aspects for detection effectiveness.

Our detection technique is sensitive to web installation routines as e.g. used by the Java or Flash update processes and thus quite often misclassified them as malware. The comparably high false positive rate of up to 7% on installer routines can be explained by the fact that, from the perspective of the employed heuristics, a web installation process in fact does similar things as a malware reloading additional payload from the internet. To cope with this problem we would thus need to accompany the corresponding detection heuristics with additional, more-specific detection patterns. Considering all analyzed samples of non-malicious processes, we achieved an aggregated false positive rate of about 1.0% for the non-quantitative case, and 1.6% if quantitative heuristics were used (see Table 2). This loss in detection precision when using quantitative heuristics can be explained by the limited specificity of quantitative heuristics in comparison to non-quantitative ones.

## 4.2 Efficiency

A crucial requirement for all malware detection and analysis approaches is their efficiency. In particular, if such an approach should be used for runtime malware detection, time efficiency is highly relevant since slow detection gives malware the change to interact with the system for some time without being stopped and thus increases the chances for a sophisticated malware to purge its traces of spread in a way that is hard to be later coped with. On the other hand, if a detection approach is to be used for forensics, space efficiency is of crucial importance to minimize to-be stored forensic data. For that reason we considered both, time and space aspects for our efficiency evaluation.

To get a realistic baseline for these efficiency evaluations we built the graphs on the logs that were also used for the goodware and malware effectiveness test. In addition we incorporated a log of a system were we tried to simulate a realistic usage behavior. This included web surfing, checking emails, writing and opening documents in Microsoft Word, listening to some music files in a media player, as well as

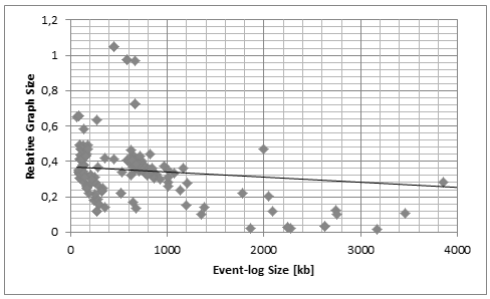



Figure 4: Scaling factor - graph to log size [kb]

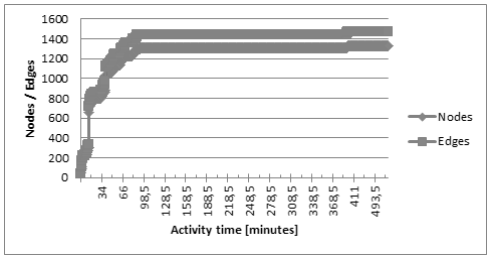



Figure 5: Graph growth - total complexity over time

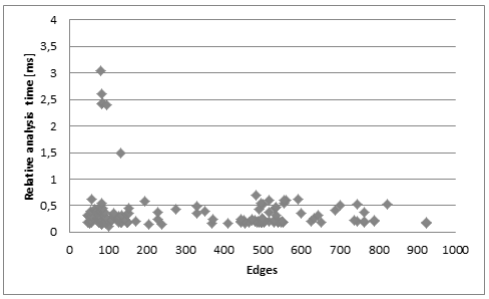



Figure 6: Relative analysis time per edge

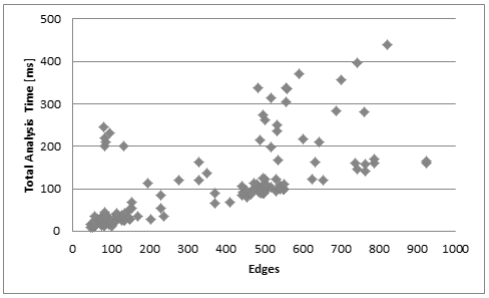



Figure 7: Absolute analysis time per edge

copying files between different folders of the file system. This log captured the behavior of this system during a period of about 9 hours.

### 4.2.1 Space Efficiency

In terms of space efficiency we first analyzed the relationship between raw log file sizes and the sizes of the corresponding generated graphs. Due to the way we built our graphs, in particular the aggregation as defined by the *update* function, we expected to see an increasing scaling factor between graph and log size for increasing log file sizes. The findings depicted in Figure 4 support this hypothesis, as we can see that while the size of a log file increases, the average relative graph size and thus the scaling factor decreases. With the exception of some outliers we can see an average scaling factor of about 40% for small logs of less than 1 MByte, going down to 10-20% for bigger log files.

These findings indicate that the amount of data that need to be stored in order to perform full-system malware detection and gives rise to optimism w.r.t. scalability.

In addition to this scaling factor analysis we also investigated the average graph growth and complexity over time. Due to our aggregation algorithms we assumed that after some time of system usage, the graph would not grow as fast as in the beginning of a system run. This is because after a while most of the connections between system entities should be present (e.g. because all necessary registry keys, files, and processes have been touched or run at least once) and future activities should only result in node or edge attribute updates.

Figure 5 supports this assumption, as we can see slow growth of graph complexity over time with a short complexity jump in the beginning. This slow growth during normal usage suggests that our approach also scales in terms of space complexity over time and thus, by construction, results in reasonably small graph sizes.

### 4.2.2 Time Efficiency

For reasoning about time efficiency we analyzed the relative time it took to match a graph node against the set of heuristics with respect to the graph size in terms of number of edges (see Figure 6). As we can see, the relative analysis time is independent of the graph size and, with the exception of some outliers, within the range of 1-2ms per node.

In absolute terms this means that the detection time is quasi-linear in the amount of edges of the graph (see Figure 7). For graphs of a maximum size of 1000 edges, which roughly corresponds to the behavior of a system with average activity over a period of about 60 minutes (see Figure 5), the complete graph analysis can be conducted in under 500ms.

With an average detection time of less than one second for a complete graph analysis, we consider our approach sufficiently fast to be usable for runtime detection. Our QDFT graphs typically only grow slowly under normal usage because of the applied aggregation steps of our model, with occasional peaks whenever new programs are started. This means that, although linear w.r.t. graph complexity, the necessary analysis time also only grows very slowly with respect to system runtime.

Note that all the heuristics presented in this paper are of a local nature (checking only a small neighborhood of a given node). Since atomic system events also impact a small portion of the graph, one does not need to re-check it entirely in each step, but it suffices to analyze the impact of the system event in a local neighborhood. Therefore we limited the efficiency analysis to graphs constructed after a few minutes of activity since the analysis times are representative for updates to bigger graphs in that time lapse. In general such a local reasoning is however not possible since interesting heuristics could include system-wide properties which force the re-verification of the complete graph.

*Monitor Overhead:* For space reasons, we do not include a dedicated discussion on the performance of the event monitor. In general, the computational overhead induced by the monitor under normal operation conditions is in the order of magnitude of about 20% and thus cannot be considered a show-stopper [31].

In sum, our measurements indicate sufficient efficiency of our approach and give rise to optimism w.r.t scalability.

# 5. RELATED WORK

As mentioned in the introduction, although still popular, signature-based malware detection is ineffective against unknown or obfuscated malware and thus can, when used in isolation, be considered insufficient to cope with modern malware. Behavior-based approaches in contrast are typically able to detect unknown malware because they focus on detecting malware by analyzing its behavior and not its persistent representation.

In the past decades, a plethora of work has been published in the broad area of behavior-based malware and intrusion detection and analysis [7]. This work can be be categorized according to several dimensions: targeted level of abstraction (e.g. host- vs. network-based), underlying detection model (misuse- vs. anomaly-based), or type of used analysis (static vs. dynamic analysis).

Because of the considered level of behavioral abstraction and system-external deployment, network based detection approaches exhibit several advantages over host-based approaches, including a lower likelihood to get disabled or circumvented by malware. These advantages, however, also come with a core limitation: By design, they work at a system rather than a process level, and therefore they cannot analyze the behavior of individual processes and thus in general provide less fine-grained behavior information than host based approaches. This in particular means that they usually cannot provide information about compromised or leaked system resources. Futhermore, network-based approaches are significantly challenged if malware uses advanced obfuscation or encryption techniques to blur or hide its network behavior [26].

Since our work focuses on host-based malware detection, we limit ourselves to review host-based behavioral malware detection approaches in the following. A seminal idea or Forrest et al. [9] in host-based malware detection through process behavior analysis is to profile benign and malicious processes through characteristic sequences of their interactions with the system, e.g., through issued system calls. The idea was later caught up by more elaborate text-mining and specifically n-gram based approaches that used machine learning to improve accuracy and precision [19, 13, 17, 25, 30, 6].

Our work differs from these approaches in two ways: First, these latter typically do not assign any explicit semantics to the used n-grams. By construction they can thus detect the presence of malware but cannot yield a deeper analysis of high-level malware behavior. Our solution, in contrast, gives a clear high-level semantics of observed malware behavior patterns. Besides detecting and pin-pointing to infections, our approach thus also establishes a basis fore more comprehensive behavioral analysis, e.g., to reason about malware intentions and attack goals. Second, approaches of the cited kind are typically sensitive to advanced behavioral obfuscation techniques like reordering system calls, randomly inserting bogus calls, or substituting calls with semantically equivalent calls. Due to the used abstraction from concrete system calls to induced data flows, which are oblivious to these obfuscation methods, our approach by construction is more robust against many obfuscation mechanisms as used by modern malware [3, 27, 33].

An orthogonal line of research on behavioral malware detection bases on the analysis of dependencies and interrelationships between activities like system call invocations of processes. After the extraction of characteristic call dependencies of different knowingly malicious processes they can be used to re-identify certain dependency patterns in unknown behavior graph samples in order to discriminate between malicious and non-malicious processes. This behavior-focused line of research can be roughly subdivided into the categories: approaches that are based on the derivation and later re-identification of potentially malicious call-dependency (sub-)graphs [16, 21, 22, 12, 18], approaches that infer high-level semantics for call-dependency graphs or sequences [5, 4, 23], and approaches that tackle the data flow aspects of potentially malicious behavior [32, 10, 14, 15].

One of the first call-graph based approaches was proposed by Kolbitsch et al. [16] who introduced the idea of deriving call-graphs from known malware samples that represent dependencies between different runtime executions of system calls, their arguments, and return values and using them as behavioral profiles for later re-identification in unknown executables. This idea was later refined by clustering and deriving near-optimal graphs to increase precision [12], or anticipate malware metamorphism [18]. Recent work in this field aimed at reducing the amount of to-be maintained behavior-graphs by mining specific invariants and paths from graphs of samples from same malware families [21, 22]. These approaches have several advantages over n-gram based ideas. Deriving and maintaining call-graph databases typically requires less sampling and storage effort than n-gram based approaches and is often more resilient to reordering or bogus call insertion obfuscation techniques.

Although we also base on the construction and analysis of behavioral graphs, we base our graph generations on quantitative data flows between system entities rather than on dependencies between single system calls. This provides better resilience against reordering and semantic obfuscation as the data flow based graph construction is less affected by shuffling and replacement of semantically equivalent calls. Also related approaches often make use of NP-complete algorithms like sub-graph isomorphism analysis which has a negative impact on the time-efficiency of these approaches. Our approach in contrast is of linear time complexity in terms of graph size. Besides quantitative aspects we furthermore differ from clustering approaches as presented by Park et. al. [21, 22] in that we construct per-system rather than per-process data flow graphs which widens the detection scope to inter-process behavior.

A seminal work of the third large pillar of behavioral malware detection was presented by Christodorescu et al. [5, 4, 12]. The basic idea of these type of approaches is to give high-level semantics to observed malware behavior. Rather than matching behavior on a purely syntactic and structural level, these approaches try to extract and re-identify characteristic behavior at a semantic level. Follow-up work further enriched this idea with formal semantics for identified malware behavior [23]. The semantic perspective of

these approaches typically leads to better resilience against more simple obfuscation like call re-ordering. This is because, although resulting in mutations of call-graphs and thus challenging normal call-graph based approaches, such obfuscation attempts do not change the semantic dimension of the malware behavior and therefore cannot easily trick semantics-based approaches.

The main commonality with this line of research is that we also base our analysis on graphs and give high-level semantics for specific of malware behavior patterns. However, in contrast to the aforementioned approaches we base the construction of our behavior graphs on the analysis of *data* rather than *control* flows. As mentioned before, this abstraction increases the resilience against advanced obfuscation like permutations between semantically equivalent functionality, which is usually not given by such approaches [5]. The data flow perspective, as abstraction from control flows, furthermore reduces the set of sub-graphs or properties we need to maintain in order to detect malicious activities, which has a positive impact on detection efficiency. Intuitively, this is because one data flow property often subsumes multiple control flow properties.

The last category of related work tackles the data flow perspective of malicious behavior. One of the most prominent examples for this family of approaches is the Panorama system of Yin et al. [32] that leverages dynamic data flow analysis for the detection of malicious activities. The basic idea of Panorama is to profile data flow behavior of malicious processes through fine-grained system-wide taint analysis and match it against manually specified data flow properties. Similarly, our approach uses system-wide data flow graphs to represent process and system behavior and match them against FOL data flow invariants. However, we make use of quantitative flow aspects to increase invariant specificity and thus detection precision. Moreover, we incorporate dedicated graph aggregation, simplification, and abstraction steps which helps to keep data structures lean and maintainable, resulting in little performance impact, whereas Panorama induces performance overhead of up to 2000% which renders it unsuitable for runtime detection purposes. We mainly differ from data dependency graph based work like the one presented by Elish et al. [8] in that we leverage quantitative flow aspects for our analysis and have a system- rather than a program-centric scope by founding our analysis on data dependencies between system entities instead of individual program points.

In sum, the main difference between our work and related contributions is that to the best of our knowledge we are the first to leverage quantitative data flow aspects for the behavior-based detection of malware. Our approach also offers more resilience to behavioral obfuscations like call reordering or semantic replacements. In addition, graph simplification allows us to optimize storage and computational efforts, resulting in a superior space and detection time efficiency.

# 6. DISCUSSION AND CONCLUSION

We have proposed a novel approach to leverage quantitative data flow analysis and graph analysis for the detection of malicious activities. Through the specification of heuristics in form of malware-specific quantitative data flow patterns, we can then identify data flow activities that relate to the presence of malware. To demonstrate the practicability of our generic model, we presented an exemplary instantiation for Microsoft Windows operating systems based on a prototype that builds on top of a user-mode monitor, capturing process activities in form of data flows induced by process calls to the Windows API.

On the basis of a proof-of-concept evaluation we showed, that our approach is able to effectively discriminate between malware and goodware with competitively high detection (up to 96%) and low false positive rates (less than 2%) compared to related approaches from literature [5, 16, 32]. These results are based on preliminary measurements and thus do not necessarily generalize due to the relatively small sample size and inherent bias in the selection of analyzed malware and goodware. For future work we thus plan to extend our evaluation to a larger set of malware and goodware samples, as well as to longer system runs to more deeply investigate graph complexity over time.

The evaluation results also indicate good space and time efficiency: the complexity of the data flow graphs turned out to only grow slowly over time for normal system usages and we observed a linear relationship between the graph complexity and the time it takes to perform a full system malware detection. The full detection time for average system activity remains under one second with an average detection overhead of less than one millisecond per graph edge. In comparison to the data-flow based malware detection approaches closest to ours [32] we achieved a better performance with comparable detection and false positive rates. These preliminary results indicate that our approach is suited for multiple malware-related application areas including runtime malware detection and analysis, as well as offline forensics, and has better time and space efficiency than most related approaches.

*Limitations:* Although using quantitative data flows as abstraction of malware activities raises the bar for malware to hide its presence through obfuscation, the quantity-based detection mechanisms are challenged if the malware obfuscates flow quantities by e.g. blurring malicious flows by scattering them over time or adding bogus data flows. We acknowledge the threat arising from the so-called base-rate fallacy [1] on the effectiveness of intrusion and malware detection approaches, including ours. According to this paradox, even a low false-positive rate of less than 2% can render an approach ineffective if malicious activities are considerably less frequent than benign ones. To cope with these problems and to further reduce false positives we plan to come up with more elaborate and thus more resilient combinations of (potentially adaptive) quantitative heuristics.

In addition, our current implementation is solely based on static data flow heuristics which only allows us to detect malware that has the specified behavioral characteristics. Although we used a wide range of behavioral descriptions of different malware families to derive our heuristics, new malware could thus simply achieve its goals in a way that we did not anticipate. To thwart this issue we intend to investigate the effectiveness of state-of-the-art machine-learning based anomaly detection on the basis of quantitative data flows.

Also the security of our prototype itself is out of the scope of this paper. Sophisticated malware could for example try to disable the event monitor to evade detection. We plan to counteract this threat by moving our monitor down the stack to the kernel layer which would give us a certain degree of tamper-resilience against user-mode malware. In sum, we

have shown that quantitative data flow analysis can be used for effective, efficient, and resilient malware detection.

*Acknowledgments* This work was done as part of the DFG's Priority Program SPP 1496 "Reliably Secure Software Systems," ref. numbers PR-1266/1-2.

# 7. REFERENCES

[1] S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, 2000.

[2] U. Bayer. *Large-Scale Dynamic Malware Analysis*. PhD thesis, Technische Universität Wien, 2009.

[3] J.-M. Borello and L. Me. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, pages 211–220, 2008.

[4] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India Software Engineering Conference*, pages 5–14, 2008.

[5] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-Aware Malware Detection. *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, 2005.

[6] G. Creech and J. Hu. A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns. *Computers, IEEE Transactions on*, pages 1–1, 2013.

[7] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, page 6, 2012.

[8] K. O. Elish, D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Workshop on Mobile Security Technologies*, 2012.

[9] S. Forrest, S. Hofmeyr, a. Somayaji, and T. Longstaff. A sense of self for Unix processes. *Proceedings of Symposium on Security and Privacy*, pages 120–128, 1996.

[10] M. Fredrikson, M. Christodorescu, J. Giffin, and S. Jhas. A declarative framework for intrusion analysis. In *Cyber Situational Awareness*, pages 179–200. 2010.

[11] M. Fredrikson, M. Christodorescu, and S. Jha. Dynamic behavior matching: A complexity analysis and new approximation algorithms. *Automated Deduction-CADE-23*, pages 252–267, 2011.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. *2010 IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[13] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st Conference on Workshop on Intrusion Detection and Network Monitoring - Volume 1*, pages 6–6, 1999.

[14] S. T. King and P. M. Chen. Backtracking intrusions. In *ACM SIGOPS Operating Systems Review*, pages 223–236, 2003.

[15] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th Conference on USENIX Security Symposium*, 2006.

[16] C. Kolbitsch and P. Comparetti. Effective and Efficient Malware Detection at the End Host. *USENIX*, 2009.

[17] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: Using system-centric models for malware protection. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 399–412, 2010.

[18] J. Lee, K. Jeong, and H. Lee. Detecting metamorphic malwares using code graphs. *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

[19] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56, 1997.

[20] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *Security Privacy, IEEE*, 2011.

[21] Y. Park and D. Reeves. Deriving common malware behavior through graph clustering. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[22] Y. Park, D. S. Reeves, and M. Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 2013.

[23] M. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, pages 1–12, 2007.

[24] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Information Security*, pages 1–18. 2007.

[25] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, pages 639–668, 2011.

[26] C. Rossow, C. Dietrich, and H. Bos. Large-scale analysis of malware downloaders. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 42–61. 2013.

[27] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[28] Symantec. Malware database, Nov. 2013.

[29] P. Szor. *The Art of Computer Virus Research and Defense*. 2005.

[30] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 67–76, 2013.

[31] T. Wüchner and A. Pretschner. Data loss prevention based on data-driven usage control. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 151–160, Nov 2012.

[32] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of*

*the 14th ACM Conference on Computer and Communications Security*, pages 116–127, 2007.

[33] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300, 2010.

# APPENDIX

In the following, we will use projections $!_i$ with $(x_1, \ldots, x_k)!_i = x_i$ for $1 \leq i \leq k$. We use the same notation to address a specific element $y$ in the $i$-th position of a sequence $Y$.

# A. TEMPORAL CONSISTENCY

For the accuracy of our analysis it is important that we consider paths between nodes that have a certain temporal consistency. Consider the following example. Process $P_1$ reads from file $F$ at time $t_1$. Then process $P_2$ writes to file $F$ at time $t_2 > t_1$. Then, although it is possible to find a path from $P_2$ to $P_1$ in the data-flow graph, there has not been an actual flow from $P_2$ to $P_1$, because of the order in which the events happened. We characterize this temporal consistency with the following theorem.

THEOREM 1. *Let $p = (e_1, \ldots, e_k) \in P$. A flow from $e_1!_1$ to $e_k!_2$ is possible if and only if:*

$$max_{i=1\ldots j-1}(min(\lambda(e_i, time))) < max(\lambda(e_j, time))$$

*for $0 \leq j \leq k$.*

PROOF. Clearly, there is a flow from node $n$ to node $n'$ if an only if there is connected path from $n$ to $n'$ of length $k$ and an increasing sequence of $k$ timestamps such that data is transferred through the connecting nodes. By induction on $k$, it is easy to see that if the proposition holds, it is possible to construct such a sequence. On the other hand if the proposition does not hold, this is impossible, because for some $j$, there is a predecessor minimum which is greater than the current maximum time-stamp, so that the sequence can not be increasing. □

Note that this characterization is still an approximation: consider a similar example as before, now process $P_2$ writes from file $F$ at time $t_1$. Then process $P_1$ reads from file $F$ at time $t_2 > t_1$. In this case although possible, there is no guarantee of an actual flow in an information theoretical sense from $P_2$ to $P_1$, since we do not know whether $P_1$ reads the portion of $F$ written by $P_2$

# B. FORMALIZATION OF HEURISTICS

In the following, we provide formalizations of the heuristics described in Section 3.2. These formalizations come as predicates $\phi \subseteq \overline{N} \times \mathcal{G}$ that are to be read as follows. If $\phi(n, G)$ is true for some node $n$ in a given QDFG $G$, then the heuristics corresponding to $\phi$ suggests that malware is present. This means that the evaluation must be performed for each node.

To simplify the description, we define a sub-string function to extend heuristics definitions with string comparisons: $ss : \mathcal{S} \times \mathcal{S} \rightarrow \{0, 1\}$ which evaluates to true, iff the first provided string is a sub-string of the second string parameter.

## B.1 Replication Heuristics

**Local replication** is formalized as

$$\phi_{\text{rep}_1}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet \\ (n, Type) = F \wedge (e_1!_2, type) = P \wedge ss(e_1!_2, n) \wedge \\ (e_2!_2, Type) = P \wedge (e_1, size) = (e_2, size).$$

**Network replication** is formalized as

$$\phi_{\text{rep}_2}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet \\ (n, Type) = F \wedge (e_1!_2, type) = P \wedge ss(e_1!_2, n) \wedge \\ (e_2!_2, Type) = S \wedge (e_1, size) \leq (e_2, size).$$

**Binary infection** is formalized as

$$\phi_{\text{rep}_3}(n, G) := \{(e_1, \ldots) \in paths_{from}(n) | \\ (n, Type) = P \wedge (e_1!_2, Type) = F \wedge ss(\text{“}.exe'', e_1!_2) \wedge \\ (e_1!_2, size) > 0\} \geq 2$$

**Download and Start Payload** is formalized as

$$\phi_{\text{rep}_2}(n, G) := \\ \exists p = (e_1, e_2, e_3, e_4, \ldots) \in paths_{from}(n) \bullet \\ (n, Type) = S \wedge (e_1!_2, type) = P \wedge (e_2!_2, type) = F \wedge \\ ss(\text{“}.exe'', e_2!_2) \wedge e_3!_2 = e_2!_1 \wedge (e_4!_2, type) = P \wedge \\ (e_1, size) \geq (e_2, size) = (e_3, size) = (e_4, size).$$

## B.2 Manipulation Heuristics

**Leaked Cached Internet Data** is formalized as

$$\phi_{\text{mpl}_1}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet \\ (n, type) = F \wedge (e_1!_2, type) = P \wedge \\ (ss(\text{“}Cookie'', n) \vee ss(\text{“}Temporary\ Internet\ Files'', n)) \wedge \\ (e_2!_2, type) = S \wedge (e_1, size) \leq (e_2, size).$$

**Spawn Shell** is formalized as

$$\phi_{\text{mpl}_2}(n, G) := \exists n' \in (pre(n) \cup suc(n)) \bullet \\ (n, type) = S \wedge (n', type) = P \wedge ss(\text{“}cmd.exe'', n').$$

**Deploy System Driver** is formalized as

$$\phi_{\text{rep}_3}(n, G) := \exists p = (e_1, e_2, \ldots) \in paths_{from}(n) \bullet \\ (n, Type) = P \wedge (e_1!_2, type) = F \wedge ss(\text{“}.sys'', e_1!_2) \wedge \\ (e_2!_2, Type) = P \wedge (e_1, size) = (e_2, size) \wedge \\ (ss(\text{“}wdreg.exe'', e_2!_2) \vee ss(\text{“}sc.exe'', e_2!_2)).$$

## B.3 Quantitative Heuristics

**Single-hop Data Amount Entropy** is formalized as

$$\phi_{\text{qnt}_1}(n, G) := (n, type) = P \wedge \\ NE(\bigcup_{e \in (in(n) \cup out(n))} \frac{(e, size)}{\sum_{e' \in (in(n) \cup out(n))} (e', size)}) \geq 0.8$$

with the normalized entropy NE defined as:

$$NE(S) := \frac{-\sum_{s_i \in S} s_i * \log(s_i)}{\log(|S|)}.$$

**Registry Call Ratio** is formalized as

$$\phi_{\text{qnt}_2}(n, G) := (n, type) = P \wedge \\ \frac{|\{e | e \in in(n) \wedge (e!_1, type) = R\}|}{|in(n)|} \geq 0.8.$$