DAVAST: Data-centric System Level Activity Visualization

Tobias Wüchner, Alexander Pretschner, Martín Ochoa
Technische Universität München
Garching bei München, Germany
{wuechner,pretschn,ochoa}@cs.tum.edu

ABSTRACT

Host-based intrusion detection systems need to be complemented by analysis tools that help understand if malware or attackers have indeed intruded, what they have done, and what the consequences are. We present a tool that visualizes system activities as data flow graphs: nodes are operating system entities such as processes, files, and sockets; edges are data flows between the nodes. Pattern matching identifies structures that correspond to (suspected) malicious and (suspected) normal behaviors. Matches are highlighted in slices of the data flow graph. As a proof of concept, we show how email worm attacks, drive-by downloads, and data leakage are detected, visualized, and analyzed.

1. INTRODUCTION

Because malware is increasingly using design-time and run-time obfuscation techniques, signature-based detection approaches are widely believed to require complementation by behavior analysis. We consider host-based anomaly and malware detection on the grounds of quantitative data flow graphs (QDFGs, [10]). In this approach, the runtime behavior of a system is monitored and represented as a data flow graph. Nodes are OS-level sources and sinks such as files, registry entries, sockets, and processes. As system calls—in our implementation, Windows API calls—are executed, data is transferred in-between the system entities, leading to directed edges in-between the nodes. For discrimination purposes, we found it useful to record not just the fact *that* data has flowed [11] but also *how much* data has flowed [4, 10]. This information can easily be deduced from the system calls. For instance, writing to a file or a socket always involves a number of bytes or blocks. Since reading from or writing to a file or a socket usually involves multiple respective system calls, it turned out to be useful to aggregate the amounts of data on the edges: rather than creating one edge per write system call from process p to file f, we create one edge for all subsequent writes from p to f. These behavior graphs nonetheless grow quickly, and in order to ensure scalability, we concentrate on slices of these graphs that correspond to a fixed but arbitrary time span.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VizSec'14, November 10 2014, Paris, France.
Copyright 2014 ACM 978-1-4503-2826-5/14/11 ...$15.00.
http://dx.doi.org/10.1145/2671491.2671499.

These graphs can be analyzed both at runtime and after the fact, by analyzing logs. To automate the analysis, we specify rules that describe typical patterns for both malware and "normal" behaviors. These rules are both described explicitly using a special domain-specific language (e.g., if a process spawns multiple children, and each of these children does the same, this is suspicious behavior) and, using data mining techniques, inferred from large bodies of malware behavior. Using these two sets of rules, we perform pattern matching by checking graph monomorphisms. This yields a certain confidence if one specific system run—or rather, the projection of that run to the chosen time window—exhibits one or more malware or normal behavior patterns.

Not only because of the problem of false positives, intrusion detection alone is usually not sufficient. Instead, system analysts are also interested in understanding what the (supposed) malware has done, and how critical this is. This is why we propose to represent the above mentioned QDFGs visually and allow the analyst to browse through the graph, step forward and backward in time, see which patterns have been identified, and where which data may have flowed as the consequence of an intrusion. Our system allows to detect multiple benign and malign activities at the same time. It represents the findings in a purpose-specific multi-view setting. The analyst can choose a tree-view (decomposition view of inferred high-level activities such as browsing or sending email); a timeline view (highly abstracted big picture of benign vs. malign activities over time without hierarchical decomposition); a statistics view (highly aggregated view of frequency/relation of benign and malicious activities over time); and a graph view which allows for the in-depth analysis of observed activities (follow flows over time and perform reachability analyses).

Problem: As security analysts, we need to perform runtime and ex-post detection, tracking, and analysis of benign and malign activities at the host-level over time. This is relevant for monitoring attackers or malware on compromised systems; for honeypots; for sandboxes; for online forensics.

Solution: We represent system activities (interaction of processes with system entities using the Windows API) as aggregated QDFGs and identify malign and benign substructures. We visualize information relevant to the security analyst as graphs, activity trees, and timelines.

Contributions: Because it does not take the perspective of (quantitative) data flows, related work does not allow to track behavior and interactivity over time and/or to scope

and filter relevant flows. The direct abstraction of raw events into aggregated data flows increases data sparsity and thus performance. Moreover, inferring abstract high-level activities through identification of characteristic sub-graphs allows filtering noise, or uninteresting behavior. Our multi-view representation allows purpose-specific filtering of information. In contrast to most related work we do not depend on system emulation or virtualization: our approach is lean as we only need to install a monitor service that automatically sends events to an analysis and visualization component.

2. RELATED WORK

Malware analysis and intrusion detection can be performed from a network perspective by analyzing network traffic or from a host perspective by analyzing interactions within a system. Graph-based approaches have been widely investigated for visualizing network security aspects [3, 2], but have been barely researched in the context of host-level security mechanisms. In the following we thus concentrate on work that is closest to ours: visualization techniques for host-based behavioral (dynamic) intrusion analysis.

Xia et al. [12] propose a visualization technique based on system monitoring information that depicts the behavior of processes in a period of time as trees, including spawned processes and interaction with other system entities such as files read. If an entity such as a file is accessed by several processes, the file will appear in all subtrees of the respective processes. To help human analysts interpreting the results, they propose a technique that allows to highlight entities to which there is a potential data flow from a declared tainted source. Different from our approach, they do not consider quantities in the flows, and there is no visual aggregation in the form of graphs, which in our experience is crucial for human interpretation. Furthermore, in contrast to our work, they do not perform any kind of activity inference to annotate low-level interactions with high-level semantics.

Trinius et al. [7] focus exclusively on samples already classified as malware. They run the samples in a sandbox and depict their behavior using treemaps and thread graphs which makes their classification and understanding more amenable. Treemaps is basically a visual representation of the proportion of certain API calls executed by the malware (such as create socket or load dll). Thread graphs give a chronologically representation of the operations performed. In contrast, our approach logs entire system runs, and, based on a weighted graph representation of data flows, classifies interaction sequences as malign or benign.

Quist et al. [6] use dynamic analysis to monitor the control flow of single processes and then visualize their results by coloring nodes depending on the context of the original program and thickening edges according to the number of executions of that particular sub-path. They differ from our work as their scope is on data flows within processes (memory accesses), whereas our work has a system wide view.

Other works such as [5, 9] use visualization techniques for the binaries themselves. This visualization however is not meant to be used by humans, but rather help during detection and declassification by comparing bitmaps.

To the best of our knowledge, our work is the first to consider the visualization of quantitative data flows between system entities in the form of weighted graphs, together with heuristics that interpret the context of flows to help human analysts interpret the presented results.

3. PRELIMINARIES

We study malicious processes by analyzing QDFGs that represent a system's data flow activities within a specified time frame. Data flow activities in QDFGs abstract from the behavior and interaction of all processes and resources in a system. We use a generic quantitative data flow graph model [10] that captures all aggregated and quantified data flows (edges) between interesting entities (nodes) in a system, such as processes, files, or sockets. The edges are directed and timestamped and intuitively reflect transfers of specific data quantities.

QDFGs are incrementally built on the basis of data flow relevant system events, e.g. functions that read data from a file or write data to a socket. At runtime, events are intercepted by monitors which interpret the data flow semantics and perform corresponding graph updates such as the creation or modification of nodes or edges. One QDFG describes all data flow activities of a system that happened during a specific time interval.

We simplify QDFGs at creation time by aggregating semantically related data flows between pairs of nodes through summation of the data amount on the respective edges rather than creating one distinct edge per event. Space limitations prevent a formal definition; see [10] for the details.

Instantiation. To instantiate the abstract QDFG model for a specific system context, we map it to concrete resources and events in this environment. In this paper, the execution environment is that of typical Windows operating systems.

The following system resources are relevant for both, malign and benign data flow behavior: *Processes* interact with all other relevant system entities in a way that they are either sources or sinks of flows from or to other *Registry*, *Socket* or *Process* nodes. In the examples of Section 4.2.4, process nodes will accordingly be labeled by P, file nodes by F, socket nodes by S, and registry nodes by R.

We also need to map all data flow relevant *events*. These are all Windows API functions that lead to a data flow between the above system entities. This includes functions to interact with file system resources like *ReadFile* or *WriteFile* to functions to send or receive data to or from a socket like the Winsock *recv* and *send* functions. For details, see [10].

Activity and Malware Detection. We do not only want to model low-level interactions as data flows, but also to give more high-level semantics to sequences of flows and thus to sub-graphs. We achieve this by specifying data flow patterns, i.e., characteristic sub-graphs that are known to match high-level benign activities like browsing or emailing; or malign activities such as an email worm, or a drive-by malware infection. These patterns are either defined manually by human security experts, or automatically extracted by mining sampled graphs that represent characteristic activities.

This results in a repository with patterns for benign and malign activities that are interesting from the security analyst's perspective. We then make use of a modified version of the the VF2 algorithm [1] which checks monomorphisms between (sub-)graphs. This allows us to detect patterns in unknown QDFGs. If patterns match, we annotate subgraphs of new QDFGs with the respective high-level activity semantics. In this way, we can trace activities through different layers of abstraction as a basis for all subsequent visualization steps.

4. APPROACH

Our approach is based on two pillars: i) the interception of system or specific API call activities in a system and their interpretation as data flows, aggregated in form of data flow graphs, and the re-identification of known benign and malign data flow patterns in these graphs; ii) a multi-view representation of the captured system activities, enriched and annotated with context information and inferred high-level interpretations of low-level system call activities on the basis of defined data flow patterns.

We see good arguments that our data centric system-wide view on system interactions is more intuitive to human analysts than approaches that focus on raw uninterpreted system calls without context information. By abstracting from concrete system calls we can achieve a significant reduction of data complexity and are able to filter activities that are more likely to pertain to relevant system activities from less relevant ones. The system-wide data flow perspective furthermore not only allows for identifying known (benign or malign) activity patterns or activity anomalies. It also enables analysts to perform more far-reaching ex-post analyses by tracking the flow of data from or to suspicious processes to e.g. assess the extent potential damages or estimate the amount of potentially leaked data.

Finally, our multi-view visualization concept aims at representing all relevant information that can be obtained from the inference and analysis steps in a purpose-specific way. This allows us to provide the right level of information granularity and abstraction for different analysis use cases. Our tool can exclusively present a coarse-grained timeline view of all captured activities for a quick overview on the system state; or a detailed data flow graph based visualization of all activities during a defined time interval for an in-depth analysis of detected malign and benign activities.

4.1 Architecture

The DAVAST system consists of two subsystem, an event monitoring component deployed at a to be monitored system, and the actual DAVAST analysis and visualization system running on the analyst's side (Figure 1).

The event monitor component intercepts all relevant Windows API calls issued by any process in the monitored system and then either forwards these events to the DAVAST system in real-time, or stores them locally for offline processing in form of event trace logs (more details in [11]).

In consequence, DAVAST supports two operation modes: In the online mode, the DAVAST system continuously receives and processes events sent from a monitored system; and analyzes and visualizes them on the fly. In the offline mode, DAVAST allows to load previously recorded event traces for ex-post analyses.

We implemented the two distinct operation modes to anticipate the requirements of several typical security analysis tasks. The online mode is well-suited for online forensic purposes like analyzing the activities on a infected system, in order to assess whether it has been infected by a malware or compromised by an attacker and to analyze attacker or malware behavior.

The offline mode is better suited for use cases where permanent human observation and analysis of system activities is not necessary. Examples include settings where the monitor component is permanently installed on workstations that are prone to be attacked or infected by malware, servers that offer important services or host sensitive data, or dedicated honeypots. In such cases, the permanently installed monitor can be configured to continuously store all intercepted system calls of the monitored systems at a secured location. Later on, they can be batch-processed by DAVAST's analysis engine in defined time intervals. If the engine then detects suspicious data flow patterns in the graphs built from these logs, they are flagged for further analysis by a human analyst. The analyst can then use the DAVAST visualizer; in order to manually verify the reported suspicion, to eliminate false positives, or to perform further-reaching analysis steps. In this way, the involvement of expensive human analysts can be reduced. At the same time, it is possible to refine the automated pattern-detection-based analysis with a more intelligent and flexible manual analysis by an experienced human operator.

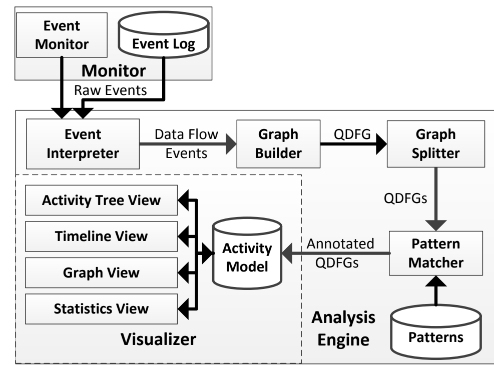


Figure 1: Architecture

The architecture of the DAVAST system is depicted in Figure 1. Depending on the operation mode, raw system call events are received from the *Event Monitor* component, or they are loaded from a recorded *Event Log*. These events are then interpreted by the *Event Interpreter* with respect to their data flow semantics and forwarded to the *Graph Builder* that aggregates them into QDFG (Section 3).

Despite event aggregation and graph simplification steps, real-world QDFGs quickly grow large with several thousand nodes end edges. For efficient processing and visualization that is comprehensible to human operators, DAVAST uses a *Graph Splitter* component. It splits the full graph into approximately equal-sized graph slices that capture the activities of a configurable time interval.

After splitting, the QDFG slices are forwarded to the *Pattern Matcher* component. This component tries to match each slice against a list of loaded predefined data flow *Patterns* that pertain to known benign or malign activities, as discussed in Section 3. This step is at the core our approach, as it annotates matching sub-graphs of the QDFG slices with inferred high-level activities. Enriching QDFGs with high-level semantics allows to trace between different levels of abstraction for detected activities, from a high-level activity description down to corresponding low-level sub-graphs.

The annotated QDFG slices are then pushed to the *Activity Model*, along with additional context information like corresponding time intervals, hierarchical relations between different (sub-)activities, level of confidence of the predicted activity, and a flag whether it is considered malign or benign. Finally, the different views of the *Visualizer* component visualize projections of the data stored in the *Activity Model*.

4.2 Visualization and Interaction

The purpose of DAVAST's visualization concept is to offer the human operator just the right level of granularity and abstraction that he needs for a specific security analysis task. To achieve this, the *Visualizer* of DAVAST is implemented as a plug-in system. Plug-ins can access the *Activity Model* and even manipulate it to some extent. The current DAVAST prototype comprises four distinct views that aim at different typical security analysis tasks and goals. Because of defined interfaces and a clear Model-View-Controller architecture, it is comparably easy to extend DAVAST with additional views.

The views themselves are projections of the *Activity Model* data as they usually only load specific types and dimensions of information from the model. The *Activity-Tree View* for example uses the inferred high-level activity semantics and the information about respective activity relations and hierarchies and association of activities to time intervals from the model. On the other hand, it ignores the structure of the respective QDFG slices. In contrast, the *Timeline View* uses information about top-level activities, and ignores hierarchical relations.

Due to the shared data model, all views are connected to each other. This allows traceability through different representations and abstractions of activities. For instance, if a user clicks on one top-level activity in the *Timeline View*, the corresponding top-level event in the *Activity-Tree View* is highlighted to visually connect multiple representations of the same concept. Correspondingly, if the user double-clicks on an activity in the *Activity-Tree View*, the system opens a *Graph View* window that visualizes the graph slice that corresponds to the selected activity.

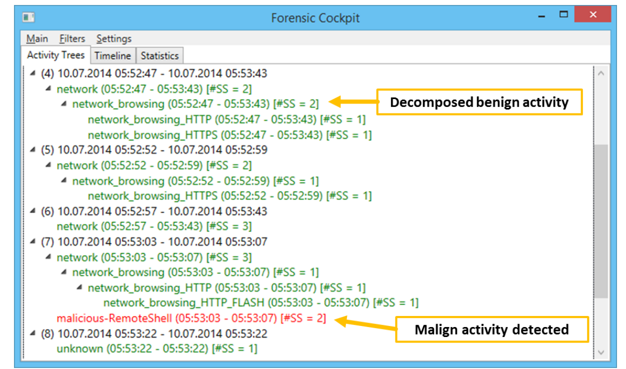


Figure 2: Activity-Tree View

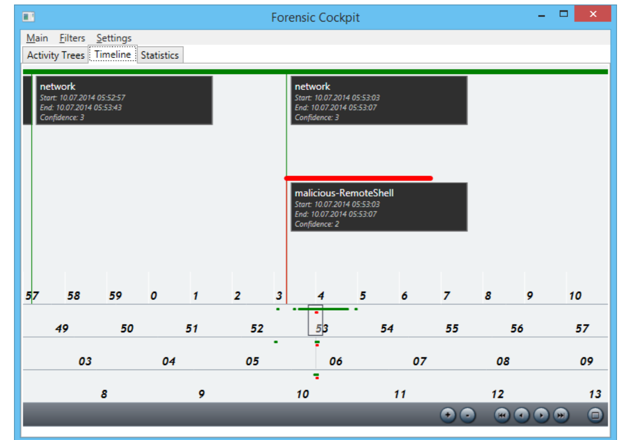


Figure 3: Timeline View

4.2.1 *Activity Tree View*

The purpose of the *Activity-Tree View* is to visualize the hierarchical relation between different activities. It depicts the list of time intervals, stored in the *Activity Model*, each denoted by a time interval number, a start, and an end time, and populates each time interval item with the activities associated with this interval. Instead of representing all contained activities in a flat way, the *Activity-Tree View* hierarchically nests related activities according to the activity hierarchy information stored in the *Activity Model*.

Figure 1 shows an example of such a decomposition where DAVAST detected a *network* activity within time interval 4. The corresponding more specific sub-activities indicate that this *network* activity in fact was a *browsing* activity, further refined into *HTTP* and *HTTPS* activities.

DAVAST colors activities in green if they pertain to benign activities, and in red if they relate to known malicious ones. For each activity, we show a prediction confidence level that represents the number of occurrences of the corresponding activity data flow pattern in the associated QDFG slice. If DAVAST identified multiple distinct activities during one time interval, this confidence number allows the human analyst to reason about the dominance and temporal proportion of one specific activity with respect to the other activities within the same time frame.

The *Activity-Tree View* concept is useful in situations where a human analysts wants to get a coarse-grained overview of all captured activities, with the possibility to get more detailed information about specific activities on demand.

By default, DAVAST initially only shows the top-level activities within each time frame and hides all sub-activities. These can be decomposed step-by-step along the hierarchy by expanding the respective sub-activities. Moreover, as mentioned before, the analyst can always open the respective *Graph View* to analyze the corresponding QDFG slice and thus the lowest level of interaction.

An example for such a setting is a scenario where the human analysts wants to analyze the system behavior of a system that he suspects to be compromised, but does not exactly know how and when it was attacked. Even if no known malign pattern matched one of the QDFG slices, the analyst can step through the time intervals and decompose interesting activities to isolate and then further analyze potential infection or attack entry points.

4.2.2 *Timeline View*

The purpose of the *Timeline View* is to give a human analyst a quick overview on the “healthiness” of a system.

In contrast to the *Activity-Tree View*, the *Timeline View* does not show hierarchical relationships between detected activities. However, it allows a more convenient way to browse through the timeline and get an overview of the proportion of potentially benign and malign activities over time.

To that end, the *Timeline View* contains a view where top-level events stored in the *Activity Model* are visualized in a chronological order. Four timeline band controls allow the user to browse through the timeline with various degrees of precision. The top-most timeline band depicts and con-

trols the *seconds* of the timeline, the one below depicts the *minutes*, followed by the bands relating to *hours* and *days*.

Similar to the *Activity-Tree View*, the system highlights potentially malicious activities in red, and benign ones in green. A colored bar visualizes the absolute event duration, independent of time intervals, whereas the *Activity-Tree View* only depicts the relative duration of an activity within the considered slice of the graph.

Multiple activities during the same period of time are stacked on each other. The activity with the highest confidence level or dominance is placed on top. Activities with lower dominance are placed below, in a descending order.

A zooming function allows the user to control how many activities are displayed on the view panel at one time to e.g. get a coarse birds-eye view on the overall system health.

Figure 3 shows a *Timeline View* of a potentially infected system. In this example we can see that both a benign *network* and a malign *RemoteShell* activity started on 10.07.2014 05:53:03, and that the malign activity lasted for about 7 seconds, whereas the benign *network* activity was ongoing.

4.2.3 Statistics View

The intention of the *Statistics View* is to provide highly aggregated statistical information about all activities detected within the monitored period of time. This allows human analysts to quickly identify anomalies in the activity profile that are likely to correlate with unwanted system behavior, as a consequence of attacks or infections.

In a nutshell, this view provides information to analysts to perform manual visual anomaly detection. This extends DAVAST's "hard" pattern matching functionality, based on fixed data flow patterns, with "soft" human anomaly detection capabilities, primarily based on intuition.

To this end, the *Statistics View* visualizes the proportion between the different detected activities over time (usually, more than one activity is taking place at a time). The x-axis represents the time interval number as defined in the *Activity Model*. The y-axis depicts the relative proportion of one event type in comparison to all other since the beginning of the monitoring period. More precisely, the proportion is defined as the number of occurrences of events assigned to a given activity type divided by the absolute number of occurrences of all events in the observation period.

Figure 4 depicts the statistics of a typical browsing session. The browsing session starts with an activity peak that reflects a database reading access of the browser. This is the browser initialization phase where configuration data is loaded from a local file-based database. Subsequently, we can see that the browsing session constantly consists of about 45% retrieval of pictures via HTTP, of about 30% retrieval of JavaScript code, and of less than 5% HTTPS communication and interaction with the local browser database.

Due to the highly aggregated way of representing activity information, this views is a good starting point for forensic investigations. In a forensic scenario, a security analyst can use the *Statistics View* to spot anomalies in the operation of a monitored system with respect to isolated time intervals that potentially contain attacker or malware activity. Examples of such anomalies include irregularly dominant HTTPS activity of an enterprise workstation during non-working lunch time or other non-working hours. During these hours, it can with some confidence be ruled out that

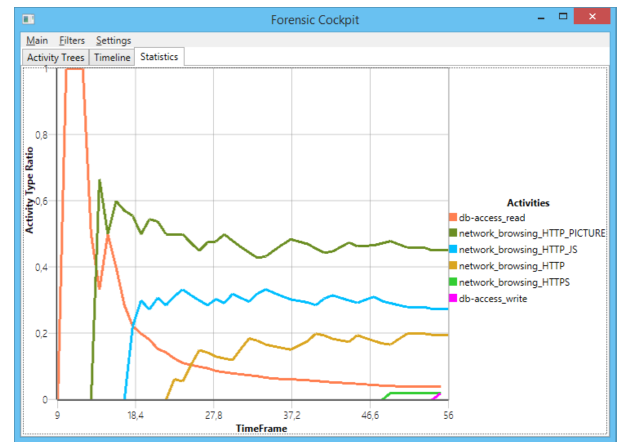


Figure 4: Statistics View

the activities are caused by legitimate usage.

Even if no explicitly defined malicious data flow pattern matched the underlying QDFGs, such an initial suspicion, as consequence of a visually identified behavioral anomaly, can still be the starting point for more detailed investigations using the *Activity-Tree View* or the *Graph View*.

4.2.4 Graph View

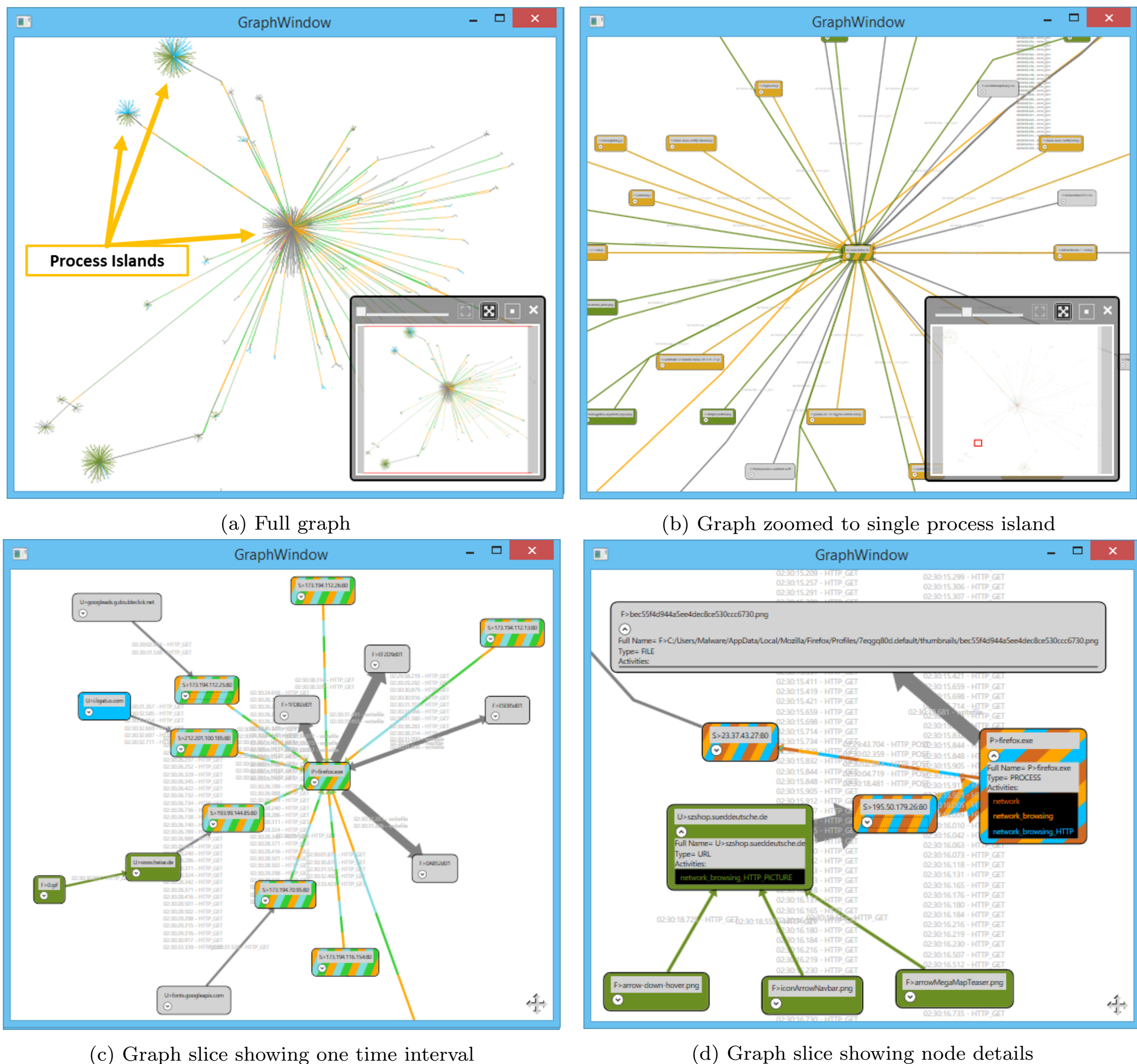
The *Graph View* is DAVAST's most central view as it provides a visualization of the core QDFG-based model that captures all system interactions during the monitored period of time. The *Graph View* provides the highest level of detail and most-fine grained visualization to the human analyst.

Its goal is to provide information about all low-level interactions in a detailed way, but thanks to the data flow abstraction, in a concise and comprehensible manner. To access the *Graph View*, the user can either double-click on a (sub-) activity in the *Activity-Tree View* to only visualize the graph slice that pertains to a specific time interval (Fig. 5c), or open the full graph view that visualizes the QDFG of the entire monitored time span (Fig. 5a).

The reason for this distinction is that QDFGs can become huge. In many cases their sheer size renders them useless to be analyzed by humans if they are visualized in entirety. While the full graph view is good to get an overview on the entire system interaction landscape and to quickly track interactions across time interval boundaries, the time interval graph view is more suited to analyze specific interactions within a bounded period of time. These time interval boundaries can be enlarged and shrinked at runtime. This gives users full control on the amount of visualized data flow interactions within one window.

For the same reason, the *Graph View* features a special zooming control in the lower right corner of the graph view window that allows to conveniently zoom in and out within a full graph or a graph slice(Fig. 5a). For higher zooming degrees an embedded mini-map that always shows the entire graph context with the currently zoomed part of the graph highlighted limits the risk of the user from getting lost while browsing through the graph (Fig. 5b).

To give the graph visualization an intuitive appearance, we use a modified *linlog* layouting algorithm that arranges the nodes and edges in a way that the graph builds so-called *process islands*. These refer to small circle-shaped



(a) Full graph

(b) Graph zoomed to single process island

(c) Graph slice showing one time interval

(d) Graph slice showing node details

Figure 5: Graph View

sub-graphs of closely connected nodes that pertain to the interactions of one specific process. In these sub-graphs the node in the center of the circle represents the currently acting process. Passive resources that this process interacts with, including files, sockets, or registry keys, are located on the periphery of the circle.

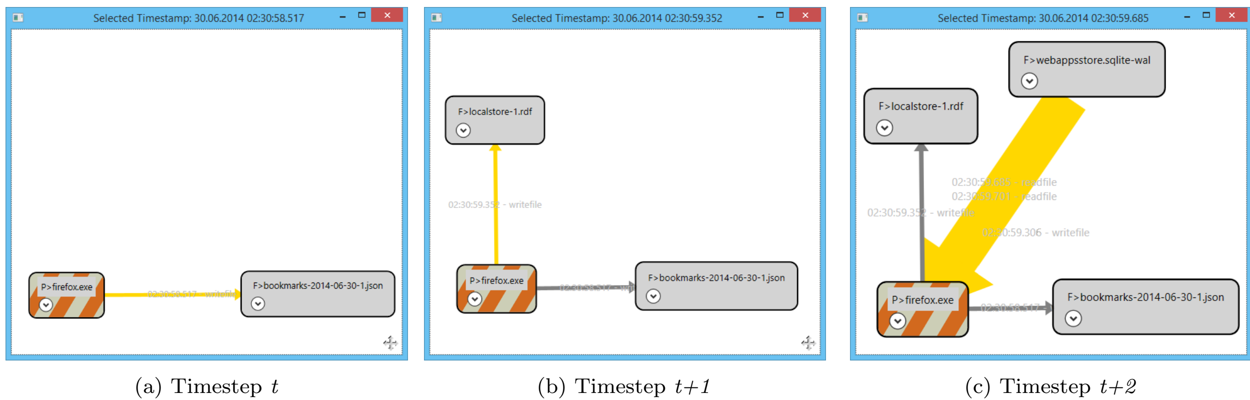
We visualize the amount of data by the edge's thickness. This gives the human analyst additional means to assess the importance and consequences of a specific sequence of interactions on the basis of transferred data amounts. In forensic use-cases, for instance, this helps the analyst assess the dimension of a potential data leakage. Edges are also labeled with the time stamps of the underlying data flow events that were aggregated into that edge.

As can be seen in Figure 5a, the different *process islands* are sparsely interconnected. This is a result from processes comparably rarely exchanging data with each other, while they more often interact with distinct sets of passive system entities. If processes exchange data with each other, this rarely happens through direct memory access between communication partners, but much more often via shared files, or socket communication. In the graph, this is represented by intermediate socket or file nodes between pairs of communicating processes.

To relate high-level activities to the corresponding subgraphs in the QDFG visualization, DAVAST assigns a unique color to each activity and paints the respective nodes and edges with this color, if they are part of the corresponding data flow pattern (Fig. 5c). In cases where an edge or node relates to multiple activity patterns, it is painted with a stripe pattern containing all relevant activity colors.

For the special case where one or more nodes and edges are detected to be associated with a known malign activity pattern, these edges and nodes are colored in red. All other nodes and edges are visually faded by ignoring their associated activity colors, and painted in a light gray instead. This directly draws the attention of the human operator to the interactions that are considered malicious (Fig. 8).

For presentation purposes, all node labels are kept as short as possible to prevent overly large nodes. This means that



(a) Timestep t (b) Timestep $t+1$ (c) Timestep $t+2$

Figure 6: Graph View - Chronologically browsing through interactions in TimeMachine mode

for file nodes, instead of directly showing the fully qualified name of the file, only the file name without its path is depicted within the node.

When interested in all details of a specific node, users can expand a details panel (Fig. 5d), which contains all available context information associated with the node, including a colored activity legend to clarify the color-activity mapping.

To further increase the usability and comprehensibility of the *Graph View* the visualization in a purpose-driven way, DAVAST features extensive filtering capabilities to e.g. filter certain node or edge types from the visualization, or to assign visual tags to single nodes that are persistently maintained throughout all graph slices to help to track and quickly identify nodes in-between different slices.

As additional means to track flows of data across individual graph slices, DAVAST includes a reachability analysis feature. For each node in a graph slice, a user can trigger a reachability analysis, which generates a new graph that only contains nodes and edges that are directly or indirectly reachable by the selected source node, under consideration of the inherent temporal information associated with the graph edges. This analysis can also be conducted in backward mode where the new graph contains all nodes and edges that can directly or indirectly reach the selected target node.

The reachability analysis usually reduces graph complexity quite a lot as it filters all nodes that are irrelevant for a specific sequence of flows and thus eases comprehensibility of the visualization. We consider this to be a useful tool for forensic purposes whenever one e.g. wants to asses the worst-case impact of a potentially malicious process (forward analysis), or wants to isolate potential sources of a suspected infected file (backward analysis).

Finally, DAVAST's *Graph View* also includes a so-called *TimeMachine* mode where users can browse through the temporal dimension of the graph in a step-by-step manner. This is done via a keyboard-controlled step-by-step construction of the currently active graph slice.

If the *TimeMachine* mode is active, the *Graph View* initially only shows the edge and connected nodes with the smallest time stamp in the corresponding time interval. By repeatedly pushing the arrow keys, the user can add or remove additional edges at the granularity of single time stamps. The current edge is colored gold, which allows to conveniently follow the sequence of interactions over time.

Figure 6 shows a brief excerpt of a graph slice in time machine mode that visualizes the interaction of a Firefox browser process with other system entities.

5. APPLICATION

In the following we present three scenarios where our approach can be useful to understand malicious behavior. In particular we show, how the visualization supports manual root-cause analysis or verification of reported alarms.

The examples are also shown in a demo video provided at `https://www22.in.tum.de/fileadmin/demos/activity_monitor/ActivityMonitor_Demo.mp4`.

Drive-by Malware Infection. An increasing threat is the proliferation of so-called drive-by malware infection attacks. To evaluate the usefulness of our approach, we locally installed a malicious web application exploiting a Flash vulnerability, resulting in a drive-by infection. The matched pattern corresponds to the process `iexplorer.exe` binding a shell (`cmd.exe`) to it, which is considered as malicious. This is detected by our approach, which allows analysts to understand the infection process as depicted in Fig. 7.

Email Worm Infection. In this scenario we open a malicious compressed attachment with Thunderbird. We then uncompress the file and execute it, resulting in an infection. As depicted in Fig. 8, we see that the malicious process replicates itself, which is detected by our library of malicious behaviour patterns. The replication pattern checks if process creates child processes or executable files of roughly the same size of the originator. A human analyst can then understand the root cause and the steps before the infection, since the pattern highlights the flow of data from Thunderbird until the self-replication. In the example, process node `invoice.exe` creates two new processes named `cpsdv.exe`.

Data Leakage. For forensics purposes, we consider the following scenario: an insider or attacker tries to copy big amounts of sensitive data to a remote server. If such transfers only rarely happen, which likely is the case for cases of data theft, a typical *Statistics View* for such a scenario would likely look as depicted in Figure 9. While the ratios of most of the benign activities are almost constant, we can clearly see two spikes for the *HTTP_Upload* activity pattern. We consider such a profile typical for cases of irregular data leakage and easy to spot by human analysts. Such an initial suspicion based on the *Statistics View* can then be a starting point for further forensic investigations, for instance by exploring the related *Graph View*.

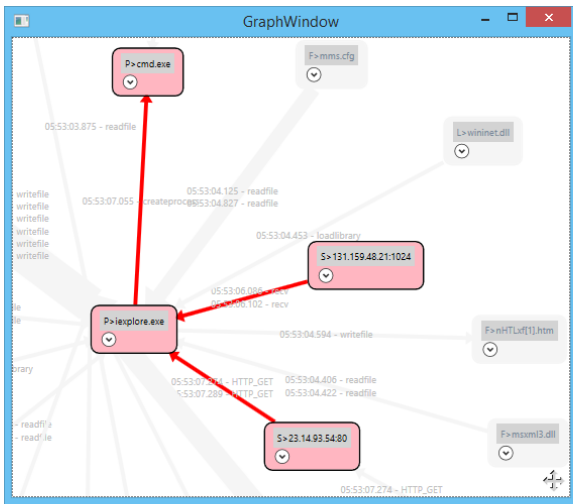


Figure 7: Drive-by Malware Infection

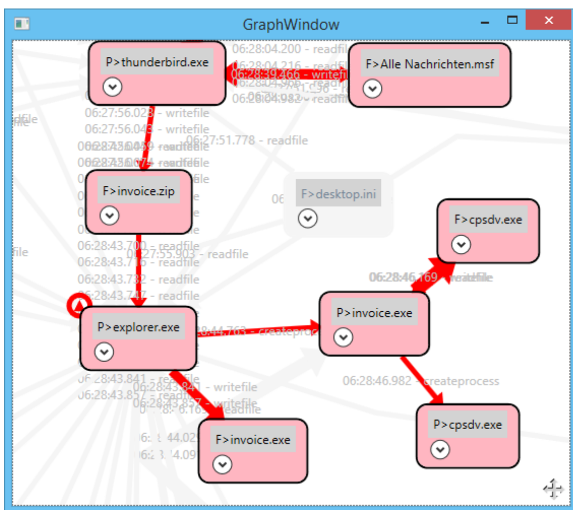


Figure 8: Email Worm Infection

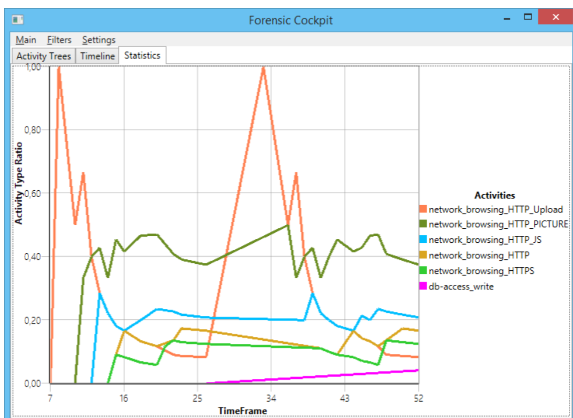


Figure 9: Data Leakage

6. DISCUSSION AND CONCLUSION

We have presented DAVAST, a visualization approach to help security analysts understand potentially malign activity. We provide different graphical views that consolidate data on system-wide data flow activity. This data is used to detect and further investigate potentially malicious behavior based on patterns. Identified malign behavior is visually highlighted to ease human analysis.

Our monitor implies an execution overhead of about 20% [11]. The analysis of a graph slice containing 1000 edges (which corresponds to a typical system usage of about 1h) is achieved under 1 second [10]. The visualization of such a graph slice takes between 2 and 5 seconds. We take this as first evidence for our intuition that our approach scales because of graph slicing, simplification, and aggregation.

We are currently extending our work in three ways: a) by improving the detection strategy, for instance by mining malicious patterns which can result in a more accurate analysis, b) by improving the visualization based on feedback by security experts, which we do in cooperation with an industry partner, and c) by improving the graph view through usage of advanced layout and filtering strategies as e.g. mentioned in [8].

7. REFERENCES

[1] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Trans. on Pattern Analysis and Machine Intelligence*, pages 1367–1372, 2004.

[2] E. Glatz. Visualizing host traffic through graphs. VizSec '10, pages 58–63, 2010.

[3] Q. Liao, A. Striegel, and N. Chawla. Visualizing graph dynamics and similarity for enterprise network security and management. VizSec '10, pages 34–45, 2010.

[4] E. Lovat, J. Oudinet, and A. Pretschner. On quantitative dynamic data flow tracking. In *CODASPY '14*, pages 211–222, 2014.

[5] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *VizSec '11*, page 4, 2011.

[6] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In *VizSec '09*, pages 27–32, 2009.

[7] P. Trinius, T. Holz, J. Gobel, and F. C. Freiling. Visual analysis of malware behavior using treemaps and thread graphs. In *VizSec '09*, pages 33–38, 2009.

[8] F. van Ham and A. Perer. Search, show context, expand on demand: Supporting large graph exploration with degree-of-interest. *Transactions on Visualization and Computer Graphics*, 15(6):953–960, 2009.

[9] Y. Wu and R. H. Yap. Experiments with malware visualization. In *DIMVA '13*, pages 123–133. 2013.

[10] T. Wüchner, M. Ochoa, and A. Pretschner. Malware detection with quantitative data flow graphs. In *ASIACCS '14*, pages 271–282, 2014.

[11] T. Wüchner and A. Pretschner. Data loss prevention based on data-driven usage control. In *ISSRE '12*, pages 151–160, 2012.

[12] Y. Xia, K. Fairbanks, and H. Owen. Visual analysis of program flow data with data propagation. In *VizSec '08*, pages 26–35. 2008.