PyTrigger: A System to Trigger & Extract User-Activated Malware Behavior

Dan Fleck, Arnur Tokhtabayev, Alex Alarif, Angelos Stavrou
George Mason University
{dfleck,atokhtab,aalarif,astavrou}@gmu.edu

Tomas Nykodym
Binghamton University
tnykody1@binghamton.edu

***Abstract*—We introduce PyTrigger, a dynamic malware analysis system that automatically exercises a malware binary extracting its behavioral profile even when specific user activity or input is required. To accomplish this, we developed a novel user activity record and playback framework and a new behavior extraction approach. Unlike existing research, the activity recording and playback includes the context of every object in addition to traditional keyboard and mouse actions. The addition of the context makes the playback more accurate and avoids dependencies and pitfalls that come with pure mouse and keyboard replay. Moreover, playback can become more efficient by condensing common activities into a single action. After playback, PyTrigger analyzes the system trace using a combination of multiple states and behavior differencing to accurately extract the malware behavior and user triggered behavior from the complete system trace log. We present the algorithms, architecture and evaluate the PyTrigger prototype using 3994 real malware samples. Results and analysis are presented showing PyTrigger extracts additional behavior in 21% of the samples.**

I. Introduction

Malware has been increasing in both sophistication and breadth in the past years. Today's malware authors are frequently government sponsored and have a wealth of experience and tools available [1]–[3]. To combat this trend, the research community has been creating automated malware analysis tools. The tools can be broken into two categories – dynamic and static analyzers. Static analysis tools examine the malware binary to extract information without actually running the tools. Dynamic analysis tools attempt to execute the malware in a controlled environment to monitor its behavior and effects in the underlying system. Each approach has different advantages and disadvantages, and frequently malware analysts use a combination of both static and dynamic analysis. Popular examples of these tools are [2], [4]–[9].

In this paper, we present a framework that attempts to expand the behavioral coverage of dynamic analysis tools beyond the current state-of-the-art. Indeed, malware research [2], [10], [11] has clearly revealed that many sophisticated malware binaries are only triggered by specific user activity. In the simplest of examples, a malicious keylogger scans the system for user keystrokes before starting to transmit the collected data to a file or over the network. More advanced malware samples constantly query window titles and browser URLs for activity to target specific websites (banking sites, PayPal, eBay, etc.). To expose malware functionality that is triggered or stimulated by user activities, we designed and implemented a novel dynamic analysis system called PyTrigger. PyTrigger mimics user activity to collect and distil from noise a more complete malware behavior profile than current dynamic analysis tools. Over a large corpus of malware our results show that 21% have behaviors triggered by user actions.

We had to overcome multiple challenges to achieve our results. The first challenge was extracting and differentiating malware behavior, triggered malware behavior (i.e. behavior caused by user stimulation) and typical system events. The PyTrigger system uses a new event differencing approach presented in section II-C2. Additionally, the system runs the samples starting from multiple initial states to accurately extract system noise from the event traces.

The second challenge we had to overcome was accurately reproducing user activities when running the malware. Our requirements for recording and playback of user events were to enable collection from a wide variety of users, combine them into a single activity trace, and play it back efficiently. While many recording and playback systems exist for testing software, none met our requirements. To collect information from many users across many Windows environments we had to record and replay context in addition to typical mouse and keyboard events. We define the context of the windows as the internal state of the widgets (e.g. which checkbox is checked, values in text fields, etc...). As screen resolutions and icon positions change simple click recording does not accurately replay a user's intent. Thus, we record and playback the context in addition to simple keyboard and mouse actions. This ensures that the object on the screen is modified appropriately regardless of position, resolution or default values. For example, if a user has specific history in their web browser, autocomplete can be used to fill in a URL. However, when replayed on our system (which isn't the system where user recording took place) the resulting URL will be different. By recording with context our replay tool can ensure the URL values match. Additionally, by creating our custom recording and playback tool we can combine multiple events into a more efficient representation. For example, combining multiple "open browser, go to URL, close browser" sequences into a sequence which only opens the browser once and loads each URL in turn or if multiple users go to the same website (e.g. Facebook) the replay will enter the site a single time.

Using our custom event extraction algorithm combined with our recording and playback tool we implemented the full

PyTrigger system to evaluate thousands of real malware samples. Additionally, we created a GUI tool an analyst could use to verify and investigate the results of the automated analysis. To summarize, the three primary contributions presented in this paper are:

1) A novel event differencing algorithm that uses multiple starting states and an event matching routine to accurately extract malware behavior and user triggered malware behavior from typical system events.
2) A new event recording and playback system that records and plays back context in addition to mouse and keyboard events. The resulting system reliably replays the user's intended activities in different environments and intelligently condenses activity events when possible.
3) A full implementation of the PyTrigger system and analysis tools which were used in experiments to process several thousand real malware samples.

The remainder of this paper is organized as follows. Section II provides details about the system architecture and algorithms created. Section III presents results of our experiments and analysis. Section IV presents a case study comparing our system to others. Section V discusses other similar work and Section VI concludes the paper.

II. System Description

The PyTrigger system has two major systems – the recording and playback system and the behavior analysis system. The recording is done through a significantly modified version of the Ranorex framework [12]. Our modifications enable us to record the context which is not possible in the original Ranorex product. Once recorded, the user traces are condensed and combined into XML files suitable for playback in our Ranorex player. Similar modifications were required in the player to enable playback of the context-aware recordings. Once complete, the user traces are passed to the playback system.

The playback system accepts a corpus of malware and runs it multiple times through virtual machines. Each VM captures a system trace through the Event Tracing for Windows facility. The traces are then analyzed by the behavior analysis system to extract malware only behaviors and malware behaviors triggered by user stimulation. The results can then be viewed and explored by the analyst using the analysis GUI.

The following sections describe the subsystems in more detail.

A. User Activity Recording and Playback

In order to trigger the user activated components of a malware sample, we must accurately reproduce user activity. User activated malware may be targeting a broad class of users or a very specific niche. Our system supports either user class. For the broader "public" we can record general scripts by executing commonly targeted applications (e.g. Facebook, banking websites, PayPal, investment sites, etc...). Public malware's goal is to hit as many users as possible, thus in many cases a general script will suffice to capture this user activity. Indeed, our results presented later show a general script triggered a large number of real malware samples. Recently, some malware has targeted more specific user populations. For example Stuxnet targeted Iran's nuclear agency [13]. Other malware may be targeted to a specific business or class of user. In these cases, our recording system can be employed to generate traces from the user population. With our ability to condense the traces by combining common activities, a representative sample of user activity can be created quickly by a specific user population. In this way, malware targeting those specific users would be triggered.

To record and playback user traces from multiple computers, we must record more than mouse clicks and key presses which can change across systems. We also must record the context of the user activity. The PyTrigger recorder records the data state of every object as its changed to ensure accurate playback. This includes Window titles, values in mutable text fields, drop down choices and all other mutable widgets on the screen. During replay these values are forced into the GUI to make sure all changes and values that happened during recording are replayed correctly (see Figure 1). For example, when a user presses the down-arrow inside a browser's URL entry field, the list of values is pulled from the history of the user on that computer. When replaying on a different system, those values will be different, and any selections made from them will be inconsistent across computers. Using PyTrigger's context-aware recording, we are able to insert the correct value into the URL and load the correct webpage.

Additionally, multiple recordings can be combined by the system to remove redundancies and create a smaller set of activities that still achieves complete coverage of the users' actions.

Once the condensed set of user behaviors is generated, they can be processed by PyTrigger's activity replay sub-system to generate traces of the malware.

B. Malware Behavior Capture and Analysis

The PyTrigger system can automatically process an entire malware corpus. The system runs each malware sample in a Windows Virtual Machine (VM) and captures the results using the Event Tracing for Windows (ETW) facility. To correctly separate malware events, triggered malware events and other system events multiple runs must be performed. Each trace is generated by running a different combination of user activity and malware as shown in Table I below.

	User activity running	Malware running
Noise Trace (NT)	Y	N
Malware Only Trace (MOT)	N	Y
Triggered Trace (TT)	Y	Y

TABLE I: How each trace is generated.

Intuitively equations 1 and 2 could be used to extract malware only events and triggered events.

$$\text{Malware Behavior (MB)} = TT - NT \tag{1}$$

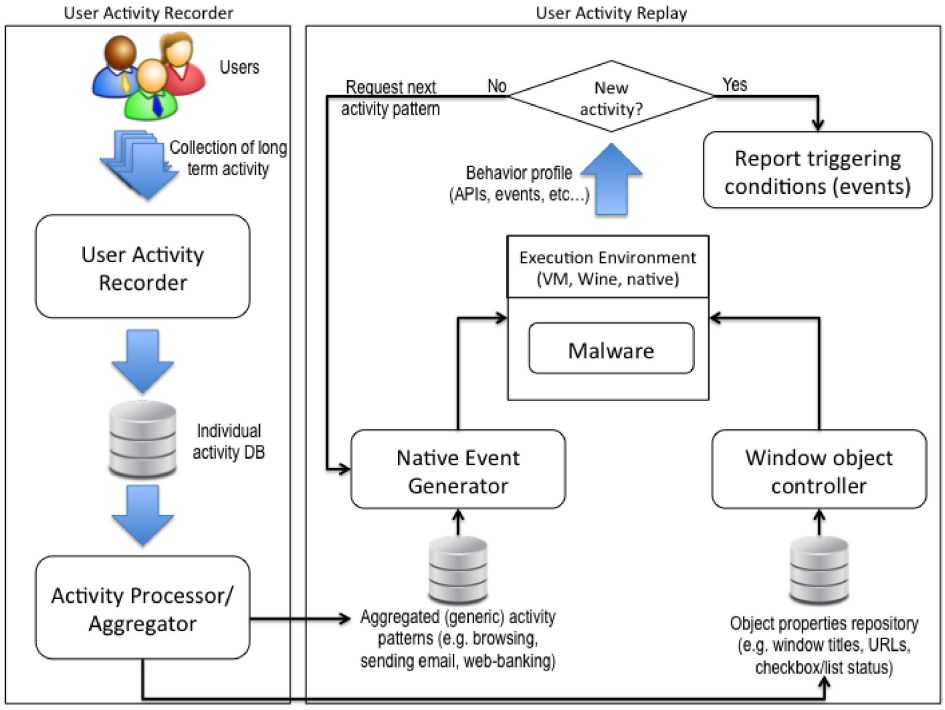

Fig. 1: Recording and Playback system overview

$$\text{Triggered Malware Behavior (TMB)} = MB - MOT \quad (2)$$

However, several problems arise in practice when using equations 1 and 2. The equations successfully remove *deterministic noise*. That is noise that appears every time we run the operating system and user event traces. However, during each run there is also *non-deterministic noise* present arising from random events that happen during processing. Non-deterministic noise can easily satisfy the condition of only being present during the Triggered Trace (TT) capture. If that happens, the noise will be incorrectly labeled as triggered behavior. To filter non-deterministic noise we run each of the trace types defined in Table I multiple times starting from various VM states. We then intersect results from each state to remove the non-deterministic noise. The modified equations are shown below where S is the set of all VM states.

$$\text{Malware Behavior (MB)} = \bigcap_{s \in S} TT_s - \bigcup_{s \in S} NT_s \quad (3)$$

$$\text{Triggered Malware Behavior (TMB)} = MB - \bigcup_{s \in S} MOT \quad (4)$$

Equations 3 and 4 are very good at extracting behavior, however they are very aggressive. For example, an event that appears in 4 out of 5 triggered traces will not survive the intersection in equation 3 and will thus be labeled noise. Similarly, if a user activity in a single state is similar to a malware activity, that activity will be removed during the subtraction operation in equation 3. To resolve this issue we introduce a state sensitivity threshold (n) defined as the number of states an activity must appear in to be counted. We then execute equations 1 and 2 individually for each state, and keep the events occurring in at least n states. By adjusting n we can increase or decrease sensitivity of the system. In section III-B we evaluate different choices for the sensitivity.

C. Algorithm Implementation

To implement the algorithm described in the previous section we must decide which events to capture, how to represent them, and how to check them for equality. The following paragraphs describe how PyTrigger implements these decisions.

1) Behavior Representation: To represent observed events in the system, we use the Event Tracing for Windows (ETW) facility with Process Monitor [14]. Each event consists of a timestamp, a process identifier (executable name and pid), an operation type, a path identifying the object of the operation, an operation outcome (success / failure) and details which contain additional information (such as requested access rights).

We consider two events equal if they have similar operation types, paths and operation outcomes. Furthermore, by using the state intersection method described previously, we restrict our behavior only to deterministic events which can be observed in several consecutive runs of the same experiment using different initial states of the OS.

2) Event Matching: To complete the event intersection and differencing we must be able to compare events for equality. Corresponding events may not match exactly due to randomized resource names or intentional polymorphism

implemented by malware. Randomized resource names occur when software uses temporary files or time-based file names. Thus, as we run samples in multiple VM states, randomized names will not have an exact match. To match these events the current system uses a very simple matching approach. The system scans the traces for randomized names before application of the behavior extraction formula. Two unpaired names (names occurring only in one of the traces) are matched if the set of operations over them matches and the names themselves are compatible. Compatibility of the names is decided depending on the type of the resource.

Network path compatibility is decided using a voting scheme by comparing the type (URL or IP), ports, and subcomponents of the IP or URL. File system and registry path compatibility is decided using a voting scheme over the following properties:

- number of parts of the path; resource names can typically be divided into parts (e.g. directory-path in case of file-system), we count the number of these parts
- if the name contains any lower or upper case characters
- if the name includes numbers
- if the name is English
- if the name includes unicode characters
- if the name appears to be random (characters equally likely)

Using the prototype we did not observe many issues with randomized names. However, our current implementation is naive and could be attacked easily. In future work we will investigate stronger approaches such as recording the call stack with the event and comparing the call site to the other trace. While the resource name may change, the call site should remain consistent (assuming user mode ASLR is disabled).

3) State Diversity: Behavior extraction effectively removes most of the events not related to malware activity (i.e. noise) from the traces. However, since we claim that the obtained set of events describes the behavior of the malware sample, any noisy event left in by formulas 3 and 4 can cause confusion and possibly misinterpretation of the sample's behavior. This is especially true for the triggered behavior, which can contain as few as 3 unique events (e.g. simple keylogger).

Noisy events can happen because most programs depend, to some extent, on the global state of the operating system. The global state includes the file-system, registry keys, network traffic and also subtle events such as context switches. The global state can be changed by other programs and cause differences in observed behavior (different set of events). Such changes in behavior may or may not be significant. For example, when malware changes a registry key causing another program to load a malicious DLL, we consider both loading of the DLL and actions performed by the DLL to be part of the malicious activity. On the other hand, if a program merely creates a new folder and another program, which is traversing the directory structure, lists the contents of this folder, then it is clearly a non-malicious side effect.

Since non-malicious side effects can be caused by OS interaction with malware artifacts, increasing the number of states will not filter such events. However, we can diversify states to broaden the definition of "normal activities" with the goal of capturing the non-malicious side effects. To diversify states we run a different set of tools as background for each state. Tools include various system snap-ins of Microsoft Management Console and some standard network tools such as netstat, tracert, net and nslookup. We also run specifically developed programs that perform an intense activity, including, file searches, process enumeration, system checks, etc. As a result, the diversified states will have activity similar to non-malicious side effects and thus the non-malicious side effects will be removed when applying equation 3.

In our experiments diversifying the states was efficient in removing most of the non-malicious side effects. However, by analyzing the initial results of our system we discovered that some events can comply with our definition of malicious (triggered) behavior without actually being caused by the malware sample nor having any significance. We call such events *persistent side effects*. They cannot be removed by our behavior extraction formula since they have the same key property as malicious (triggered) activity - they too occur if and only if malware runs together with recorded user activity in any OS state.

Persistent side effects are best exemplified by application name caching and prefetching, which are normal system activities. Application name caching occurs when Windows Explorer runs an executable, which makes the 'explorer.exe' process automatically store the application name in the registry (MuiCache key). At the same time, when a process runs any new executable, the prefetching service ('svchost.exe') will create a prefetch file, so that next time the application will load faster. Therefore, if the triggered payload executes a malicious binary in the context of Windows Explorer, then this event will be followed by a persistent side effect registry set and file create/write operations due to name caching and prefetching. Our experience shows it was often hard even for an expert to distinguish between malicious behavior and these persistent side effects. In the evaluation section we show how state diversity affects the number of irrelevant events.

III. System Evaluation

We evaluated PyTrigger on 4100 malware samples that were randomly selected from a large malware repository provided by the MD:Pro project, Cyveillance, Inc., and Google malicious URL lists. When selecting the malware samples, we tried to preserve a uniform distribution over malware families using Kaspersky Anti-Virus definitions. However, since we were primarily interested in malware which is likely to exhibit user-activated behavior, we excluded samples from two known non-interactive families (e.g. backdoors and droppers). The resulting set has 4100 samples from 35 different malware classes and 393 families.

For our experiments we collected a representative set of typical user activity related to online e-mail (Gmail, Yahoo), social networking (Facebook), online search (Google), web

banking access (HSBC), text editing, file browsing and execution (Windows Explorer). PyTrigger then executed each malware sample in each VM state and recorded the behavior. Due to efficiency of the replay system, PyTrigger can replay collected activity in approximately 200 seconds. However, to ensure robustness, we executed each sample for 300 seconds in each VM state with and without replaying user activity. Experimental results are presented below.

A. *Experimental Results*

During the experiments, PyTrigger processed all 4100 malware samples using various configurations. 11% of the samples failed to run properly for various reasons. Some of the samples disrupted our environment by terminating the behavior monitoring process or preventing simulated activity to finish. Other samples failed to run due to missing dependencies or detecting the virtual environment. We report only results of samples which ran successfully. We judged malware to run successfully if either the malicious process caused a permanent change on the system (e.g. created a file) or the extracted behavior contained at least 100 events. According to this criteria, 3994 samples ran successfully.

Our experiments indicate that PyTrigger was able to trigger and expose additional activity in 847 (**21%**) out of 3994 executed samples. Figure 2 presents the distribution of triggered malware by class name. 24 classes (out of 35) had at least one triggered sample. Figure 3 depicts the ratio of triggered samples within the classes. As expected, spyware families tend to have a higher ratio of triggered samples. However, there was no single class nor family with a triggered rate above 50%. PyTrigger triggered 52 out of 152 samples of the Trojan-Banker family giving it the highest triggering rate of 34%.

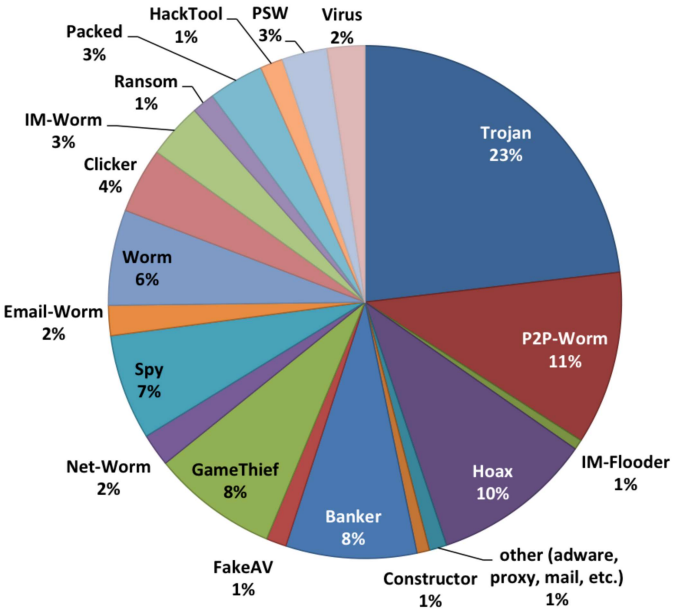

Fig. 2: Triggered malware distribution by class label.

The vast majority of malware from the Banker and Trojan-Spy families should have user-activated behavior by definition.

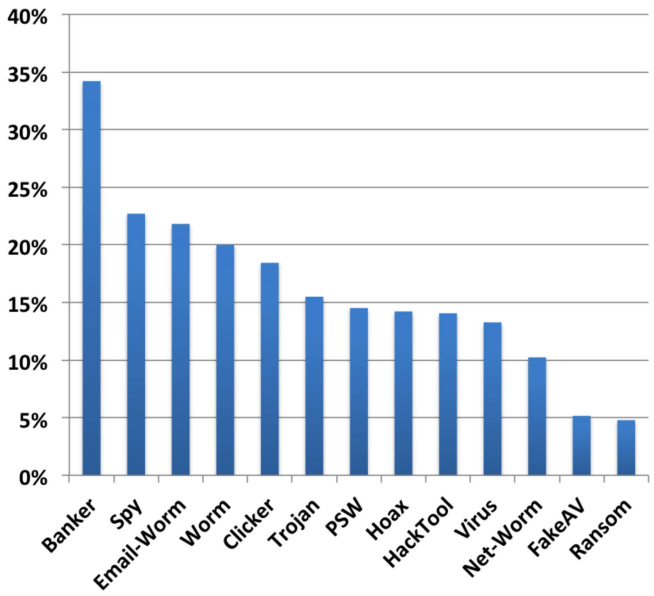

Fig. 3: Percentage of malware triggered in each class.

Their triggering rates suggest that some samples were not activated (potentially false negatives). There can be several reasons for this. First, there is no guarantee that the user-activated component of malware successfully executed (e.g., system hooks installation). Although we did not consider malware that failed to perform any activity at all, some of the samples might delay running a user activated component or crash due to an incompatible setup of the guest OS. Other samples may decide to cease execution for many reasons (e.g. detection of a virtual environment, expired dates, etc...). Second, for malware that successfully monitors user activity, we may fail to trigger because we never launch the application or visit web pages the malware is interested in. For example, by manual inspection we discovered that some samples targeted Russian websites.

1) Triggered Behavior: For each triggered sample, PyTrigger identified and extracted new triggered behavior. Table II summarizes statistics on triggered behavior with respect to system operations. In the table, the "Overall Samples" column shows the number of samples that originally performed a given operation when running without PyTrigger. The right two columns ("Triggered Samples") indicate for how many samples PyTrigger exposed an additional behavior and how many triggered samples actually delegated the behavior to other processes respectively. Table II indicates that PyTrigger activated some additional behavior in 847 samples. Our system triggered new file and registry operations in 411 and 460 samples respectively. PyTrigger also triggered process creation in 55 samples. A comparison of PyTrigger's results versus other existing malware analysis tools is presented in section IV as a case study for a Trojan.Win32.Diple.qfo sample.

During our experiments we observed various types of triggered behavior. Some of the samples performed only a few events including only registry/file reads, whereas others

Activity	Overall Samples	Triggered Samples	
		Total	Delegated
File Creation	3598	126 (3%)	69
File Reading	2763	411 (14%)	335
File Modification	3598	185 (5%)	134
Registry Querying	3496	460 (13%)	335
Registry Modification	3598	185 (5%)	134
Network Operation	258	74 (28%)	49
Process Creation	3707	55 (1.4%)	26
Total	**3994**	**847 (21%)**	551

TABLE II: Additional behavior exposed by PyTrigger

showed complex behavior such as dropping and launching an executable. By analyzing the results, we noticed that many of the triggered behavior traces were similar to each other, sometimes even identical. We believe this is due to reuse of a user activated (UA) component which can be embedded in different malware samples.

It was not always possible to understand the purpose of the observed triggered activity. However, it was often possible to identify the same component across different traces (especially thanks to default names of created resources such as "keyboar.dat"). We were able to recognize several keylogger components, spyware components for stealing user input from webpages and an example of a complex spyware component recording overall user activity including keystrokes and visited webpages. Out of the observed components, keyloggers were particularly widespread and were found to be embedded in malware samples of various families.

Moreover, we observed that other UA components are shared by malware from various families and even classes. This observation suggests that UA functionality is probably becoming a part of various types of malware and is no longer exclusive for traditional user activated malware such as keyloggers and spyware. For instance, UA functionality could be used to detect the sandbox environment, i.e., absence of an "alive" user [15] and disable the malicious behavior.

In summary, triggered malware results (Figures 2 and 3) indicate that samples of various classes and families have UA components which can be triggered by our system. While the success of PyTrigger for certain malware families (e.g. targeted malware) depends on richness of the user activity profiles, we were able to trigger more than 20% of the tested malware with relatively modest user activity profiles. Future work on PyTrigger will improve the triggering rate by adding more user activity patterns and by enriching the activity context in the object property repository (see Figure 1).

B. State Analysis

The behavior extraction formulas (3 and 4) use intersection and subtraction operations across the states. These formulas will fail to find malware or triggered behavior when a trace from a single state does not contain that behavior. However, when a malware sample does not run in a particular state due to a missing dependency or conflicting OS configuration we are unable to capture the behaviors. When this happens the missed behaviors are False Negatives (FNs). Additionally, a state may run a legitimate process that performs similar operations as the malware thus masking the malware activity as normal. In this section we analyze different state voting schemes designed to limit these problems. For example, the "4 out of 5" scheme means that only four states must agree to label a behavior thereby reducing the impact of a single state failure.

We performed the analysis by testing majority voting (i.e. "3 out of 5" and "4 out of 5") schemes that tolerate two and one conflicting states respectively. Additionally we tested hard schemes (n out of n) with all triples, quadruples and a quintuple of the five states we used. The results are presented in Figure 4. The tested voting schemes are on the Y-axis and the samples ordered by ID are on the X axis. If a scheme triggers a sample, that sample is shaded black. From the figure one can see that state 5 causes the samples outlined in blue not to trigger and state 2 causes samples outlined in red not to trigger[1] . However, if we loosen the voting scheme to "3 out of 5" then almost all samples are labeled triggered because more irrelevant system events pass through. From our analysis any hard scheme (e.g. "1,2,3", "2,4,5") does not provide enough state diversity to capture all the triggered samples. However, the "3 out of 5" majority scheme does not filter events aggressively enough, generating many False Positives (FPs). Thus, for all future experiments we have chosen the "4 out of 5" scheme to ensure we tolerate a single state providing an incorrect label while not passing an excessive number of irrelevant system events.

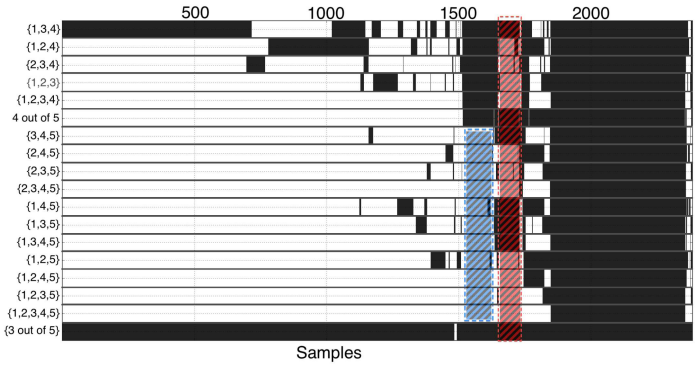

Fig. 4: Schemes triggering comparison

C. PyTrigger Delegated Event Analysis

A key advantage of PyTrigger is its ability to extract *delegated events*. These are events that are delegated by the malicious process to other legitimate processes outside the malware process chain (e.g. Explorer, svchost.exe). False positives (FPs) in our system arise when PyTrigger incorrectly labels an event outside the malware process chain as a delegated malware event. FPs only happen with delegated events because events labeled within the malware process chain are correctly attributable to the malware. It is reasonable to question any sample labeled 'triggered' by our system when the triggered events are all delegated. We call these

[1]Blue sample IDs ∼1500-1600, Red sample IDs ∼1650-1700

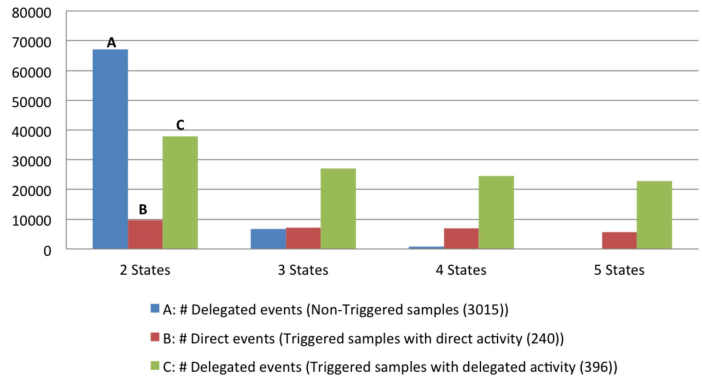

Fig. 5: Delegated activity reduction. FP events (i.e. category A) drastically reduced with increasing number of states. Over-filtering is minimal, and most good events (i.e. category B and C) survive through increasing numbers of states.

'delegated only' samples. As shown in Table II 65% of the triggered samples are delegated only samples. To validate these delegated only samples we processed them with the most restrictive ("5 out 5" voting scheme). Using that scheme there were 240 triggered samples containing some direct triggered events and 396 triggered samples containing only delegated events. We then processed all the events using equations 3 and 4 for an increasing number of states. The key property we are evaluating is the survivability of events through the intersection and differencing operation. It is highly unlikely that system noise will survive through multiple states. Thus we expect random system noise (FPs) to decrease while non-random malware events survive (TPs) as we increase the number of states being used. Figure 5 shows the results of the experiment. Three categories are graphed for each set of states. A: delegated events from non-triggered samples. B: direct events (i.e. part of the malicious process chain) from triggered samples. C: delegated events from triggered samples. As shown, non-triggered events (A) do not survive intersection and differencing as the number of states are increased. However, most direct and delegated triggered activity does survive through all states.

This means the FP rate decreases as the number of states increases making it harder for FP events to survive. The number of delegated events in the triggered behavior set (C), which is computed by formula 4, decreases rapidly from the 2 states setup to the 3 states setup; however it does not decrease much from 3 to 4 to 5 states. Such a small decrease suggests that noisy *delegated* events were mostly eliminated in the 3 states setup. We can see that the number of *direct* events does not change from the 3 state setup to the 4 state setup, but it does decrease with the 5 state setup. This means that schemes involving up to 4 states do not lose many direct events. However using more states ("5 out of 5") results in over-filtering and increases the rate of FN events.

D. Validation of triggered samples

To validate the samples we manually inspected 390 traces of a representative set of malware to determine the ground truth. We then computed the false positives (FP) rate and false negatives (FN) rate. During manual inspection of the triggered traces, we distinguished meaningful malicious activity from irrelevant system events. It was not always possible to understand the purpose of the activity flagged as triggered, especially if the activity is delegated to a legitimate process (e.g., Internet Explorer or svchost.exe). However, it was often possible to identify similar patterns across different traces.

The fact that several malware samples share an identical or very similar pattern, by itself is highly indicative that these samples are triggered. Indeed, noise (i.e. irrelevant system events) should not consistently reoccur in different malware running in different VM states. To this end, we manually classify a sample as triggered if at least one of the following conditions holds: (i) it modifies resources (files, keys) created by malware; (ii) it shares a trigger pattern with a significant percentage (0.25%) of samples.

Table III shows the false positive rate and false negative rate for different voting schemes. Columns show the best FP and FN rates that can be achieved with a specific number of states. For instance, the setup that runs 4 states can achieve very low FP (6%) using states (2, 3, 4 and 5), but with the penalty of high FN (36%). The table shows using any specific combination of states is prone to having a high false negative rate because a problem in a single state causes the intersection operation to incorrectly remove events resulting in missed malware and triggered behaviors. Conversely, the wrong set of states may pass to many system events resulting in a high false positive rate. The optimal voting scheme must maintain low FP and FN rates. The validation results presented here are consistent with results from previous sections and support using the "4 out of 5" schemes in our experimental evaluation.

E. Analysis User Interface (UI)

All the results presented have been automatically generated by PyTrigger. However, to facilitate investigation by researchers and malware analysts we have also created an Analysis UI shown in Figure 6. The timeline area shows each process with visible events on the Y-axis and a timeline of events on the X-axis. Green events are user events injected by the PyTrigger replay system. Red events are malware events triggered by user behavior and yellow events are malware events not caused by user activity. As expected, the malware events not caused by user triggering (yellow) start soon after the process begins, and the triggered malware events (red) are correlated to user behaviors (green). Using the UI an analyst can dynamically review the PyTrigger results to understand their meaning and further reduce false positives.

IV. Case Study

We selected the Trojan.Win32.Diple.qfo sample, which exposes both delegated and user activated behavior, as the subject of our case study. Log 1 presents an excerpt from the sample

	2 states		3 states		4 states		5 states	"4 out of 5"
	Min FP	Min FN	Min FP	Min FN	Min FP	Min FN	-	-
FP	13%	25%	9%	11%	6%	8%	10%	15%
FN	27%	5%	35%	16%	36%	26%	36%	13%
State Set	{3, 5}	{1,2}	{2,3,5}	{1,3,4}	{2,3,4,5}	{1,2,3,4}	{1,2,3,4,5}	-

TABLE III: False positive and false negative rates under different voting schemes

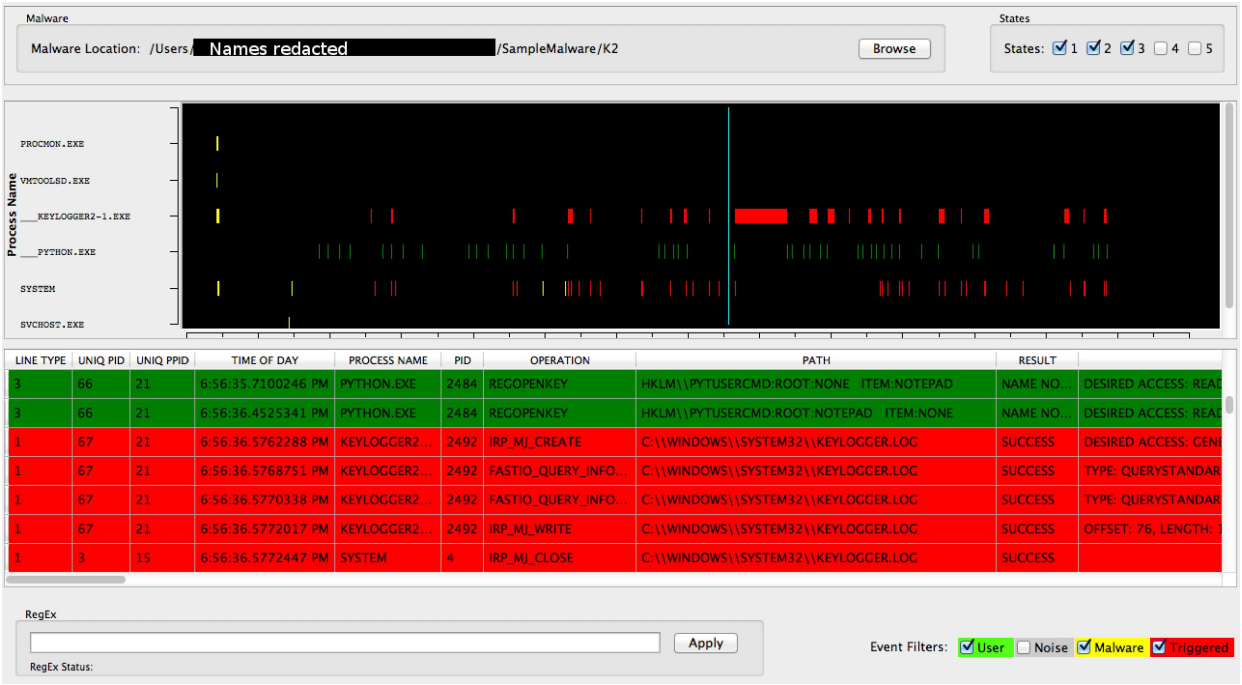

Fig. 6: Analysis tool to analyze PyTrigger results.

	Activity	PyTrigger	Anubis	CWSandbox	ThreatExpert
Direct	Registry access (query), Dll loading	**Yes**	**Yes**	**Yes**	No
	File dropping (addon.dat)	**Yes**	**Yes**	**Yes**	**Yes**
	Process creation (itself)	**Yes**	**Yes**	**Yes**	No
Delegated	Indirect dropper (svchost.exe)	**Yes**	No	No	**Yes**
	Dropper autorun	**Yes**	No	No	**Yes**
UA	Keylogging (write to log.dat)	**Yes**	No	No	No

TABLE IV: Activity discovery performance for Trojan.Win32.Diple.qfo

activity log obtained by PyTrigger. The original log included all system-wide behavior that occurred due to malware execution with a user activity. Due to space limitations in this excerpt, we omitted numerous malware operations related to registry access and DLL loading. However, we preserved the most interesting part, namely the activity performed not only by malware process but also by Windows and IE processes. Clearly, this is a case of delegated malicious activity that previously infiltrated into legitimate processes. Note that, for our approach, it does not matter how an activity was delegated; what matters is that this activity was never observed without malware running in the system.

A simple inspection of the sample Log 2 reveals that the Windows Explorer process created a malicious dropper named svchost.exe and a set registry so that the dropper is run when a user logs on. Moreover, although both Windows Explorer and IE accessed log.dat file only IE wrote data to the file.

Log 2 presents the pure triggered activity log obtained by PyTrigger. The log shows merely a few operations, nevertheless important ones (e.g. writing data to the log.dat file). Manual analysis indicated that writing to the log.dat file is indeed a part of the triggered activity, which is likely performed by a spyware component such as a keylogger.

In this case study, we compared sample activity produced by PyTrigger with behavioral profiles obtained by other popular analysis systems, such as Anubis [4], CWSandbox [6], and ThreatExpert [5]. We examined the profiles with respect to both the ability to reveal delegated behavior, which is not necessary related to user activity, and the ability to reveal user-activated (UA) behavior. The result of our analysis is presented in Table IV, which shows whether each activity was identified by a given tested analysis system.

In Table IV we distinguish between three types of malicious activities, such as direct activity committed by malicious

Log 1 Sample activity

```
8CCA.exe, 3324, Process Create,C:\experiment\8CCA.exe,SUCCESS,"PID: 3460, Command line: C:\experiment\8CCA.exe"
8CCA.exe, 3460, IRP_MJ_CREATE,C:\Documents and Settings\Administrator\Application Data\addon.dat
8CCA.exe, 3460, IRP_MJ_WRITE,C:\Documents and Settings\Administrator\Application Data\addon.dat

Explorer.EXE, 268, RegCreateKey,HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion,SUCCESS,Desired Access: Query Value
Explorer.EXE, 268, IRP_MJ_CREATE,C:\Program Files\microsoft,"Desired Access: Read ... Directory, ..., OpenResult: Created"
Explorer.EXE, 268, IRP_MJ_CREATE,C:\Program Files\microsoft\svchost.exe,"... Access: Generic Write,..., OpenResult: Created"
Explorer.EXE, 268, IRP_MJ_QUERY_VOLUME_INFORMATION,C:\experiment\8CCA.exe
Explorer.EXE, 268, IRP_MJ_SET_INFORMATION,C:\Program Files\microsoft\svchost.exe,SUCCESS,"Type: SetEndOfFileInformationFile"
Explorer.EXE, 268, IRP_MJ_WRITE,C:\Program Files\microsoft\svchost.exe,SUCCESS,"Offset: 0, Length: 65,536"
Explorer.EXE, 268, IRP_MJ_CREATE,C:\Program Files\microsoft\log.dat,NAME NOT FOUND,"Desired Access: Write Attributes, ..."
Explorer.EXE, 268, RegSetValue,HKLM\SOFTWARE\Microsoft\ ...\stubpath,"Data: C:\Program Files\microsoft\svchost.exe s"
Explorer.EXE, 268, RegSetValue,HKLM\SOFTWARE\win32\nck,SUCCESS,"Type: REG_BINARY, Data: FF 37 D4 12 ... 74 FA 93 5B 67"
Explorer.EXE, 268, RegSetValue,HKCU\Software\win32\klg,SUCCESS,"Type: REG_BINARY, Data: 01"

IEXPLORE.EXE, 3492, Load Image,C:\WINDOWS\system32\ntkrnlpa.exe,SUCCESS,"Image Base: 0xcc0000, Image Size: 0x20d000"
IEXPLORE.EXE, 3492, IRP_MJ_READ,C:\Documents and Settings\Administrator\Application Data\addon.dat, "Offset: 0, Length: 16"
IEXPLORE.EXE, 3492, RegSetValue,HKCU\Software\win32\plg1,SUCCESS,"Type: REG_BINARY, Data: EA 44 DC 02 ... 07 DA F2 35 03"
IEXPLORE.EXE, 3492, IRP_MJ_CREATE,C:\Program Files\microsoft\log.dat, "... Access: ... Read/Write, ..., OpenResult: Created"
IEXPLORE.EXE, 3492, IRP_MJ_WRITE,C:\Program Files\microsoft\log.dat,SUCCESS,"Offset: 0, Length: 42"
```

Log 2 Triggered activity

```
8:46:00.73,IEXPLORE.EXE,3492,IRP_MJ_WRITE,C:\Program Files\microsoft\log.dat,...,"Offset: 0, Length: 42"
8:46:02.92,System,4,IRP_MJ_SET_INFORMATION,C:\Program Files\microsoft\log.dat,...,EndOfFile: 42"
8:46:04.07,IEXPLORE.EXE,3492,FASTIO_WRITE,C:\Program Files\microsoft\log.dat,FAST IO DISALLOWED,"Offset: 42, Length: 41"
8:46:04.11,IEXPLORE.EXE,3492,FASTIO_WRITE,C:\Program Files\microsoft\log.dat,...,"Offset: 83, Length: 3"
```

processes, delegated activity performed by compromised legitimate processes, and user-activated behavior. It can be seen that none of the tested analysis systems were able to discover all activities identified by PyTrigger. For instance, Anubis and CWSandbox failed to identify both UA and delegated activities that apparently escaped the malicious process chain. Since ThreatExpert is based on state differences, it successfully detected delegated activity; however, it missed the non-persistent activity, such as registry query. The fact that ThreatExpert identified the delegated activity confirms that this activity is not user-activated. Moreover, as expected, none of the tested systems discovered the UA behavior of the sample. In conclusion, this case study demonstrated that PyTrigger has the potential to obtain more complete behavior than the tested systems.

V. Related Work

Dynamic malware analysis. There are several well-known malware analysis tools that obtain behavioral profiles for malware executables [4]–[6]. These tools run samples in isolated environments that are frequently based on modified VMs [16] or machine emulators [4]. In this domain, much of the research was aimed at improving *analysis granularity* [2], [8], [9] and *analysis transparency* [16]–[18]. In contrast, our work is focused on improving *analysis completeness*: extracting the complete behavior of a malware sample.

Malware triggering, behavior detection, and extraction. In [11] the authors proposed an approach to identify environment sensitivity properties in malware. While their system has a different goal than PyTrigger, they use a similar approach to behavior representation and differencing but they did not use intersection to remove the random noise and detect delegated events. In [10] the authors tackle the problem of execution stalling. Our approach is complementary to theirs because waiting for some specific user activity could be used as an effective technique for execution stalling. In both [19] and [20] researchers attempt to trigger hidden behavior in malware samples by first identifying code paths dependent on the environment and then force the execution of both options. They managed to obtain more complete behavioral profiles for several samples, however, they did not attempt to launch user activity related functionalities. Our work is also related to high interaction honeyclients [21]–[23]. Similar to us, their system needs to differentiate between legitimate noise (random behavior) and malicious behavior. They use whitelisting of certain operations which is exploitable by mimicry attacks [15].

User interaction simulation in malware analysis. The first attempt to create an automatic execution, observation and analysis framework using user input can be traced to LARIAT [24]. They build scripts to simulate background and user actions such as typing in interactive sessions in order to test certain attack scenarios. Panorama [8] and TQAna [25] simulate user activity to detect information leaks. These systems run a predefined set of user activity scripts, e.g. AutoIt scripts [26], to introduce sensitive information to target applications. Our system is different in two aspects.

PyTrigger replays user activity at both the event and data level to reproduce the user activity context, whereas AutoIt based engines used in [8] and [25] generate only events. For certain types of user activated malware, such as malicious browser helper objects, it might be sufficient to trigger information stealing functionality by simply filling certain text fields. In the general case, to trigger the event reaction routines of malware, we must replay user activity while constantly varying the activity context (UI property values) so that malware eventually encounters the context (data) it expects.

Second, Panorama and TQAna systems are mostly oriented to detect user activated information leakage. PyTrigger was designed to identify any user activated behavior. We applied a more generic approach that recognizes user activated behavior as new and then records activity in response to user interaction in appropriate context. Therefore, in contrast to [8] and [25] our user activated behavior extraction scheme is agnostic to dependencies between executed malware/OS events such as the event execution chain and information flow.

Joe sandbox [7] uses AutoIt scripts to simulate primitive user activities such as visiting web pages and filling web forms. However, it does so to activate malware that remains dormant in the case of no user interaction. Siren [27] uses user activity to detect malware which tries to blend in with legitimate network traffic. They inject user behavior with a known network profile into the system and look for differences from the expected flow. The system is similar to ours in that we also inject user activity with a known profile (i.e. noise trace) and look for the difference. However, we detect user activated behavior on the network and local machine.

Overall, most tools usually restrict the monitoring to events generated by the tested sample process chain [1]. Our experiments show that a malware can hide its activity inside other processes or by invoking other processes. Contrary to existing work, our approach can identify behavior of multi-process/partite malware that actively delegates its malicious functionality to several, individually benign processes [28].

VI. Conclusions

In this paper we presented a novel approach to extract a more comprehensive behavioral profile of malware by triggering user activated components of the malware. To do this we created a custom user activity recording and playback tool that generates a more complete and accurate user activity trace by including the context. Our approach also facilitates combining and condensing traces from many users into a single trace. We then applied our new technique to extract both malware behavior and triggered malware behavior. From these results we are able to extract malware functionality that remains hidden during typical dynamic analysis.

Using our prototype implementation called PyTrigger we tested a substantial set of 3994 real malware samples. Our results show PyTrigger was able to create a more complete behavioral profile for 21% of the tested samples. The majority of the user activated behaviors discovered are executed in hooked processes. Thus, PyTrigger's ability to detect delegated behavior was pivotal to increasing the triggering rate, especially for spyware and keyloggers. We validated our results by manually labeling a significant sample of malware showing PyTrigger labels samples with 86% accuracy and 85% precision. We also presented a case study in section IV showing PyTrigger outperforms other available tools.

In the future we will investigate ways to automatically determine likely triggering mechanisms and target PyTrigger's user activity playback to those mechanisms. We also found that user activated behavior is common across many malware classes, and frequently seems to be the result of shared components. Future work will generate an even more complete profile by exploiting common components.

References

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

[2] E. Bangerter, S. Bühlmann, and E. Kirda, "Efficient and stealthy instruction tracing and its applications in automated malware analysis: Open problems and challenges," *Open Problems in Network Security*, pp. 55–64, 2012.

[3] S. W. Brenner and A. C. Crescenzi, "State-sponsored crime: The futility of the economic espionage act," *Hous. J. Int'l L.*, vol. 28, p. 389, 2006.

[4] "Anubis: Analyzing unknown binaries," http://anubis.iseclab.org/, 2012.

[5] "Threatexpert," http://www.threatexpert.com/, 2012.

[6] "Cwsandbox," https://mwanalysis.org/, 2012.

[7] "Joe sandbox," http://www.joesecurity.org, 2012.

[8] 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*. ACM, 2007, pp. 116–127.

[9] M. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. of the 18th Annual Network and Distributed System Security Symp.*, 2011.

[10] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 285–296.

[11] M. Lindorfer, C. Kolbitsch, and P. Comparetti, "Detecting environment-sensitive malware," in *Recent Advances in Intrusion Detection*, 2011.

[12] "Ranorex: Test automation tools," http://www.ranorex.com/, 2012.

[13] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[14] M. Russinovich and B. Cogswell, "Process monitor v3.03," http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx, 2012.

[15] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, "Escape from monkey island: evading high-interaction honeyclients," *Detection of Intrusions & Malware, & Vulnerability Assessment*, pp. 124–143, 2011.

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[17] A. Nguyen *et al.*, "Mavmm: Lightweight and purpose built vmm for malware analysis," in *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE, 2009, pp. 441–450.

[18] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[19] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *IEEE Symposium on Security and Privacy*, 2007, pp. 231–245.

[20] D. Brumley *et al.*, "Automatically identifying trigger-based behavior in malware," *Botnet Detection*, pp. 65–88, 2008.

[21] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy, "A crawler-based study of spyware on the web," in *Proceedings of the 2006 Network and Distributed System Security Symposium*, 2006, pp. 17–33.

[22] Y. Wang *et al.*, "Automated web patrol with strider honeymonkeys," in *Proceedings of the 2006 Network and Distributed System Security Symposium*, 2006, pp. 35–49.

[23] "Capture - honeypot client," 2006, available from https://projects.honeynet.org/capture-hpc.

[24] L. Rossey *et al.*, "Lariat: Lincoln adaptable real-time information assurance testbed," in *Aerospace Conference Proceedings*, vol. 6, 2002.

[25] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *USENIX Annual Technical Conference*, 2007, p. 18.

[26] "Autoit v3," http://www.autoitscript.com/site/autoit/, 2012.

[27] K. Borders, X. Zhao, and A. Prakash, "Siren: Catching evasive malware (short paper)," in *Security and Privacy, 2006 IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 6 pp.–85.

[28] W. Ma, P. Duan, S. Liu, G. Gu, and J. Liu, "Shadow attacks: automatically evading system-call-behavior based malware detection," *Journal in Computer Virology*, pp. 1–13, 2011.