SysTaint: Assisting Reversing of Malicious Network Communications

Gabriele Viglianisi
Politecnico di Milano
gabriele.viglianisi@mail.polimi.it

Michele Carminati
Politecnico di Milano
michele.carminati@polimi.it

Mario Polino
Politecnico di Milano
mario.polino@polimi.it

Andrea Continella
Politecnico di Milano
acontinella@iseclab.org

Stefano Zanero
Politecnico di Milano
stefano.zanero@polimi.it

ABSTRACT

The ever-increasing number of malware samples demands for automated tools that aid the analysts in the reverse engineering of complex malicious binaries. Frequently, malware communicates over an encrypted channel with external network resources under the control of malicious actors, such as Command and Control servers that control the botnet of infected machines. Hence, a key aspect in malware analysis is uncovering and understanding the semantics of network communications.

In this paper we present *SysTaint*, a semi-automated tool that runs malware samples in a controlled environment and analyzes their execution to support the analyst in identifying the functions involved in the communication and the exchanged data.

Our evaluation on four banking Trojan samples from different families shows that *SysTaint* is able to handle and inspect encrypted network communications, obtaining useful information on the data being sent and received, on how each sample processes this data, and on the inner portions of code that deal with the data processing.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Virtualization and security**; **Software reverse engineering**;

KEYWORDS

Malware analysis; software reverse engineering; virtualization; communication protocol; botnet

ACM Reference Format:
Gabriele Viglianisi, Michele Carminati, Mario Polino, Andrea Continella, and Stefano Zanero. 2018. SysTaint: Assisting Reversing of Malicious Network Communications. In *Software Security, Protection, and Reverse Engineering Workshop (SSPREW-8), December 3–4, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3289239.3289245

SSPREW-8, December 3–4, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6096-8/18/12…$15.00
https://doi.org/10.1145/3289239.3289245

1 INTRODUCTION

In the last years, the malware criminal industry has continuously grown, with toolkits that allow cybercriminals to easily run profitable malicious campaigns (e.g., ransomware [7], banking Trojan [6]) being sold on the black market. As a result, new malicious specimens keep appearing on the Internet at high rates. According to McAfee Labs Q4 2017 Threat Report [1], 57.6 million new samples have been observed in Q3 2017. As a consequence, security researchers developed scalable automated tools to analyze large amounts of samples.

A malicious sample can either be studied statically, by analyzing its binary code, or dynamically, by running it in a controlled environment and observing its behavior. Malware, in turn, employs several techniques to evade these two analyses and to remain undetected. Automated analyses are commonly performed through *sandbox software*, which executes the sample in a controlled environment and monitors its activities, collecting data about its interactions with the operating system and highlighting malicious behaviors. There is still, however, a practical need to manually reverse engineer malware samples in order to understand their inner workings and to deal with modern evasion techniques and new malicious behaviors. This task remains hard and time-consuming.

In particular, it is hard to analyze and reverse malware samples when they communicate over an encrypted channel with a Command and Control (C&C) server [27]. The reversing of these communication functionalities is itself a major challenge, but is fundamental to gather insights about the malware families and the capabilities of the samples under analysis.

In this paper we present *SysTaint*, a semi-automated tool that allows running malware samples in a controlled environment and analyze their execution to extract information about the data-flows and the inner network-related functionalities. These information can be interactively explored to dig into the malware's behavior and quickly find the malicious code handling a given piece of data. For instance, our tool can track the data read from a file or received over the network, and help to understand how this data is processed before it is sent over the network. This allows to effectively debug, in a semi-automated fashion, malicious functionalities that rely on network events.

We implemented *SysTaint* on top of the Platform for Architecture-Neutral Dynamic Analysis (PANDA) [9], a framework that allows to record a sample execution, replay it deterministically, and study

in details the executed trace during the replay phase. *SysTaint* monitors the communication between the sample and the operating system, tracks the data-flows, logging the involved internal functions, and automatically detects the usage of encryption. By inspecting the recorded execution through *SysTaint*, a malware analyst can easily find the provenance of encrypted the data sent over the network, locate the corresponding unencrypted data and the code that deal with the data processing. Finally, we integrated *SysTaint* with the Cuckoo Sandbox [2], a widespread open source sandbox, to leverage its functionalities and to ease the deployment in existing analysis environments.

We experimentally verified *SysTaint* by analyzing four samples of banking Trojans whose behavior relies on exchanging data with a C&C. Thanks to *SysTaint*, we were able to quickly inspect the content of encrypted network communications, obtaining useful information on the data exchanged over the network, and on the malicious code processing such data.

In summary, we make the following contributions:

- We implemented *SysTaint*, a system that extracts from the (recorded) execution of a given sample information about the data-flows and the code involved in the processing of each data-flow. We implemented *SysTaint* as a set of plugins for the PANDA platform.
- We showed that *SysTaint* is able to assist analysts in reversing malicious network communications (e.g., malware exchanging data with a C&C), by identifying and extracting both the code and data involved in network activities.
- In the spirit of open science, we made *SysTaint* publicly available for the community[1].

2 BACKGROUND AND MOTIVATION

Automatically analyzing malware has become a practical necessity for security firms, which need to determine the behavior, nature, and family of a large number of malware samples to protect companies and end users. Hence, the security industry has developed highly sophisticated solutions to automatically detect if a given binary sample is malicious (e.g., antivirus software [26]) and tools to analyze the behavior of unknown software. Automated analysis solutions employ a variety of techniques that can be classified in static and dynamic. While static analysis techniques extract information by analyzing the code of the sample, dynamic analysis techniques consist in running the sample in a controlled environment to gather information on the behavior and the inner workings of a sample.

2.1 Motivation

The most common dynamic analysis solutions are based on sandbox environments and are designed to process a large number of samples. These solutions run each sample in a virtual machine, tracking and logging the interactions of the sample with the operating system and known libraries. Even though automated sandbox analysis systems are widely employed in observing the behavior of a malicious sample, they offer limited help in the reverse engineering task since they do not fully capture the behavior and inner workings of the sample. On the other hand, the existing automated reverse engineering approaches, and in particular the ones focusing on the communication protocols [5], are not easy to apply in practice due to the heavyweight instrumentation required and the lack of publicly available open source implementations. Therefore manual and multiple debugging sessions are still by large the most employed method to extract in-depth information on the behavior and inner working of a malware. Unfortunately, debugging requires the analysts a significant amount of time and expertise, especially when the malicious behavior is particularly difficult to replicate and inspect. Additionally, performing dynamic analysis on malware communicating with external servers may be difficult or impossible when the sample's C&C is not reachable [12].

2.2 Case Study

An example of a hard-to-reverse behavior is the communication with the C&C server. The communication is usually encrypted and unintelligible from the logs that a sandbox captures. Even when encryption is not employed, it may be useful, from the reversing point of view, to understand how the data the malware retrieves from the system is processed and used. In addition, the results of the analysis on the communication with the C&C (e.g., the data transmitted, the timing) depends on a series of factor (e.g., the server status, the time-span of the analysis, the connectivity) that makes the dynamic analysis a daunting task, since it requires to pause (when debugging) or significantly slow down (when employing heavyweight instrumentation [10]) the execution of the sample, possibly causing the network communication to fail [27, 28]. In fact, the reversing of the communication protocol is a key aspect for malware analysts, since it allows, for instance, to obtain useful information about botnets [32]. In this work, therefore, we want to focus on providing tools and information to speed up the analysis and reversing of the encrypted communications between the malware and the C&C servers.

2.3 Goals

The goal of this work is to develop a framework to support malware analysts in the dynamic analysis and reverse engineering of real-world malicious samples, bridging the gap between these two activities. In particular, we aim to ease the reversing of malware samples by leveraging the record-replay functionalities provided by PANDA [9] to perform semi-automated analyses and data extraction on the recording, and guide the manual reverse engineering of the malware code. We also aim to propose a practical mean of collecting useful information from dynamic analysis that can offer an alternative to manual debugging sessions and avoid the difficulties of debugging malware whose behavior may vary between executions because of external network inputs.

To achieve these goals, we want our system to be able to:

- record the execution, then replay and re-analyze it at a later time, thus removing the dependence of the analysis on the presence and behavior of external servers;
- uncover the contents of encrypted network communications;
- track the flow of the data the sample exchanges with the system and the network, locating how and where in the sample's code the data is obtained, transformed and used,

[1] https://github.com/vigliag/systaint

and focusing, in particular, on the data flows involved in the network communications.

2.4 Challenges

We present here the main challenges in the development of *SysTaint*:

Deriving semantics from low-level data. The main challenge of any reverse engineering task is to recognize high level behaviors from large amounts of low level instructions and data that appear meaningless when not considered in their context. For this reason, to present the analyst with meaningful information about what is happening in the program, it is important to automatically infer as much high-level information as possible. Solutions like PANDA work as a hypervisor, observing the execution of the programs by monitoring the instructions and the memory accesses being executed. Determining what these instructions and memory accesses represent in the context of the current program and operating system is a complex task called Virtual Machine Introspection (VMI). Despite, from the hypervisor standpoint, we have full visibility into system memory, some information that are easy to access at run-time, for example the current thread identifier, normally available through the `GetCurrentThreadId` Win32 API call, are harder to access by means of VMI, requiring knowledge of OS internals. Moreover, the recordings we analyze, provided by PANDA, do not capture information like the state of the files on disk, nor include the memory pages swapped out to disk. This makes it difficult to query the state of the system at a given time.

Timing. When studying the behavior of malware, timing plays a key role since, from the moment the sample is first executed, it can quickly start several other processes and inject itself into them. An infected process' memory layout and code can also change significantly during the analysis because of the use of packing and injection techniques. Some information may only be in memory for a certain time-window as, for example, temporary buffers containing unencrypted data, the unpacked code of the malware, or some dynamically loaded libraries. When using memory analysis or VMI to obtain information, it is important to access memory at the right points in time.

Isolating the behavior of the malware. Another challenge is distinguishing the behavior of the sample under analysis from the benign activities of the operating system and the other running processes. This is particularly evident when analyzing the network activity logs, as several unrelated communications from different processes can overlap. Moreover, malware code is often executed from inside an infected benign process (e.g., *Internet Explorer*), making it necessary to distinguish the actions of the malware from the ones of the host process.

Malware employing encryption. As part of the malware authors' efforts to hinder the analysis, the communications between malware and their command and control server are usually encrypted, either by using an encrypted channel (e.g., HTTPS), or by using an unencrypted channel and encrypted payloads. Recovering the unencrypted message is fundamental to the reverse engineering of the communication protocol, and usually involves finding and instrumenting the cryptographic functions called by the sample.

Malware employing evasion techniques. Malware try to evade the analysis by detecting some artifacts of the virtual environment or analysis software. Additionally, even a simple strategy such as waiting a long-enough time before acting, if not detected, can effectively hide the malware activities from the behavioral analysis. Unfortunately, countering these strategies is hard and is part of an ongoing cat-and-mouse game between malware authors and analysts. As a consequence, the presence of malware samples able to evade a given automated analysis must be assumed.

Performance and memory considerations. Dynamic analysis can have a significant CPU and RAM overhead. This is the case with taint tracking, a technique to extract dataflow information, which requires instrumenting almost every assembly instruction and potentially a large amount of memory to keep track of the tainted memory locations. When dynamic analysis is used to collect data, disk space is also a concern, and a trade-off between storage and the amount of details to record must be found. In fact, some malware samples produce a large amount of recorded activity as part of their normal operation – for example a ransomware encrypting numerous files – while others perform a huge amount of unrelated operations to produce too much activity to be recorded and analyzed, and evade sandbox analysis.

3 APPROACH OVERVIEW

SysTaint is a semi-automated tool that allows running malware in a controlled environment and analyze their execution to extract information about the data-flows and the sample's inner functions. These information can be interactively explored to dig into the malware behavior and quickly find the functions handling a given piece of data. For instance, our tool can track the data read from a file or received over the network, and help to understand how this data is processed and used. Similarly, this tool can be used to discover what data a malware sample is sending through the network and how it was produced and manipulated. By leveraging record-replay analysis solutions, extracting from the recording useful information that are usually obtained through manual debugging sessions, detecting encryption, and reconstructing the data-flows we aim to address the challenges in the study of malware whose execution relies on network communications with external servers.

From a high level perspective, *SysTaint* collects in-depth data about a subset of the functions called by the malware sample and the data they handled, and uses system call tracing, taint analysis and cryptographic functions detection to build an interactive data-flow graph. Our approach, represented in Figure 1, can be summarized in five steps:

(1) **Sample execution recording**. The activity of the malware is recorded by running it in a virtual machine, leveraging the record-replay capabilities of *PANDA* [9]. This process can be performed either manually or through an automated analysis system.
(2) **Activities of interest selection**. The analyst selects some malware behavior to inspect, using an interaction with the operating system as starting point. For example, a given API call or network exchange.
(3) **Preliminary analyses**. The tool performs some preliminary analyses on the recording: first, the identification of the

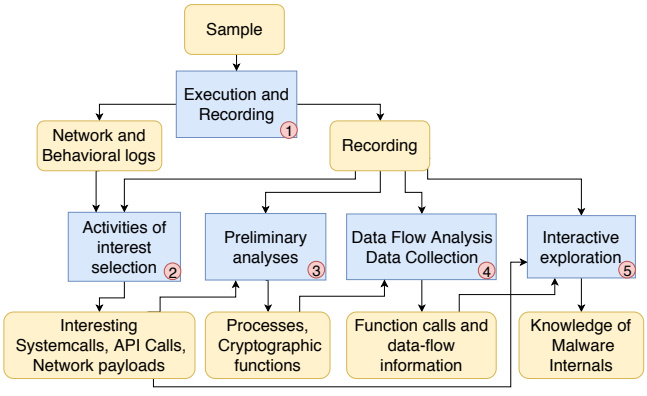

Figure 1: Overview of *SysTaint* approach. In blue the main analysis phases, in yellow the intermediate data

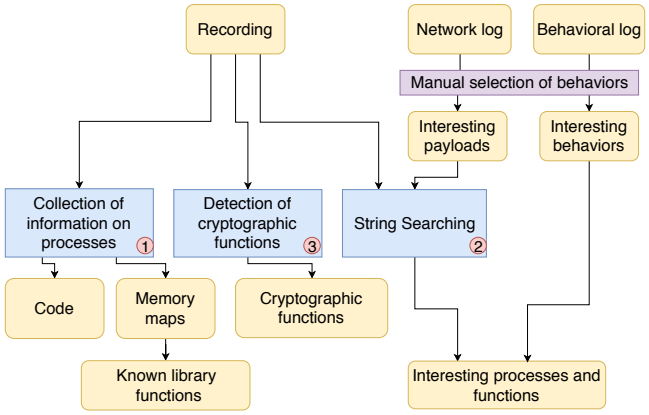

Figure 2: Overview of the the preliminary analyses

processes involved in the selected malware behaviors, then, the detection of the usage of cryptographic and compression functions in these processes.

(4) **Data-flow analysis and data collection**. The tool runs a more heavyweight analysis, performing the data-flow analysis and collecting the input and output data of a subset of the internal functions of the sample.

(5) **Interactive data exploration**. The analyst interactively explores the malware execution by querying the collected data.

3.1 Sample Execution Recording

In this step, we execute a malware sample in a virtual machine for a given amount of time and leverage the record-replay capabilities of *PANDA* to obtain a recording of the execution. In the remaining steps of our approach, we will work on the recording obtained in this step, extracting the data we need by deterministically replaying the recording with added instrumentation.

We configured an existing sandbox software to use *PANDA* as hypervisor, so to automatically obtain, during the sandbox's analysis, a recording of the execution that can be used for a later in-depth analysis and reverse engineering process. Doing so, we can also take advantage of the existing analysis infrastructure, its reports, and anti-evasion countermeasures.

In contrast to existing sandbox software, since the recording includes all the processes in the system, we do not need to make any real-time decision on which processes to instrument, and we delay the identification of malicious processes to the later *preliminary analyses phase*. This is an advantage since we do not rely on the automated and real-time detection of the mechanisms a malicious sample may use to create new processes, which is prone to fail and often excludes interesting processes from the analysis.

3.2 Activities of Interest Selection

The second step requires the analyst to manually identify some interactions of the sample with the operating system to use as starting point to study the sample's inner workings. These interactions are used in the *preliminary analyses phase* to identify the processes to include in the analysis, and later as a starting point for the *interactive data exploration*.

The analyst identifies these interactions by inspecting the network log and a behavioral log, containing the system and API calls performed by the sample, which are either provided by the sandbox or extracted from the recording. For instance, the analyst could select from the network log some encrypted data the malware sent through the Internet, to determine its unencrypted contents and provenance, or he or she could identify some system calls reading data from the Windows registry to find out how this data is used.

3.3 Preliminary Analyses

The preliminary analyses phase consists in three sub-steps, represented in Figure 2:

(1) **Collection of information on the processes** and their memory maps, to later display annotated memory addresses and obtain the addresses of known functions in Windows's APIs.

(2) **String searching** to find usages of some previously identified interesting data. This is useful to find the processes of interest, and gives insights on the functions processing the identified data as-is.

(3) **Detection of cryptographic functions** usage to easily locate them and their outputs during later phases.

3.3.1 Collection of information on the processes. In this phase, we inspect the virtual machine's memory to automatically collect information about every process. For each process, we replay the recording until a point in time when the libraries and malware code are loaded in memory, for instance in correspondence with a process' last instruction. Then, we use the Rekall [3] framework to collect:

- information and contents of the pages in the process' address space, each with boundaries, permissions, and mapped file (if any). This information is obtained by parsing the *VAD tree* [8];
- the list of stacks and heaps;
- a list of the functions exported by all the DLLs in the process' address space.

This data is useful in the later phases to recognize the use of known library functions, to obtain and display information about memory locations, and to inspect the code at a given address.

3.3.2 String searching. String searching is a technique that allows to quickly find the processes and functions dealing with some particular data. It consists in monitoring memory accesses and scanning for the occurrences of some given string of data – for example a given piece of ciphertext extracted from the network logs. From the matching memory accesses it is possible to find the code processing a given string and, examining the access patterns, infer whether the data is being read in one or multiple function calls and whether a function is simply copying the data to a different location or transforming it. *SysTaint*'s string searching is based on [10] and is rooted around the concept of *tap points*, points in a program's execution identified by process ID, program counter, and caller address. The data being read and written at each tap point is seen as a stream of bytes, and is separately monitored for the occurrence of the searched strings. While this approach is fast and efficient and can reveal the presence of repeated accesses to the same data inside of a function, it may miss some matches when the data is accessed across different tap points or not read in order. These cases can be covered employing per-function-call buffering of the data read and written: when a function call returns, the buffers it has read or written are scanned for the searched strings.

3.3.3 Detection of cryptographic functions usage. Detecting if a function performs cryptography, compression or encoding (we will use the term "cryptography" for brevity) allows us, during the data collection phase, to track its outputs, by tainting them with a unique label. The detection of cryptographic calls nodes in the data-flow graph allows the analyst to quickly locate their inputs and outputs and to better track the data-flow. As described in section 6, the literature contains several approaches to perform cryptographic function detection. We evaluated two approaches, namely the heuristic based on the ratio of arithmetic operations [5], for its simplicity, and a custom heuristic based on several per-function metrics, detailed in the next paragraphs. In our preliminary tests (see Appendix A), our custom heuristic outperformed the one based on the ratio of arithmetic operations, identifying a superset of the cryptographic functions. We thus decided to employ it exclusively for our analyses.

It is worth noting that, since we work on a recording and know the ciphertext in advance, it is also possible to locate the functions processing the ciphertext, and by extension the plaintext, with the previously described string searching techniques, by searching for functions writing the ciphertext to memory.

In order to apply the heuristics, we replay the execution collecting several per-function-call metrics:

- the size of the function (in basic blocks);
- the total number of executed basic blocks;
- the presence of loops (inferred from the previous two metrics);
- the number of arithmetic and total instructions executed;
- the size, entropy, count of ASCII characters of each read and written buffer.

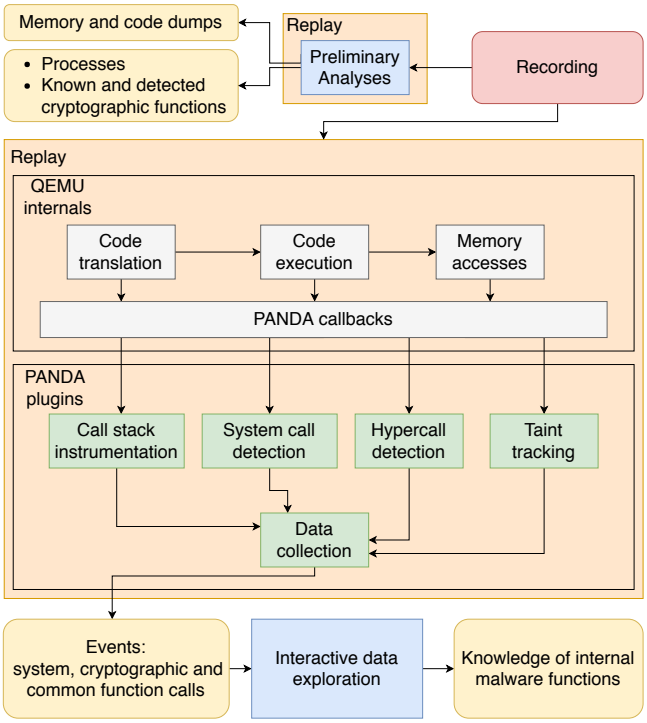

Figure 3: Overview of the data collection phase

Custom heuristic. Our custom heuristic aims to detect both cryptographic and compression functions on the basis of the following criteria:

- cryptographic/compression primitives read or write high entropy data;
- the call tree having as root a cryptographic/compression primitive is shallow;
- cryptographic/compression primitives spend most of the time in loops.

The criteria were iteratively selected evaluating the results of the sample analysis.

The above heuristic to identify cryptographic primitives is first applied to all the individual function calls. Then, a function is marked as "cryptographic" if at most two of all its function calls fail to satisfy the heuristic (i.e., if more than two function calls do not satisfy the heuristic, the function is not marked as "cryptographic"). This allows discarding functions like `memcpy` which may happen to copy high-entropy data. We then inspect the callers of the identified primitives in order to find higher-level functions that, for instance, call the identified primitives in a loop. We go through each detected primitive P, marking its caller function C as cryptographic if P is only called by C and the call trees having as roots the instances of C are also shallow. Unfortunately, since our heuristics work at function-level granularity, they cannot detect the use of smaller obfuscation primitives that have been inlined in the body of a larger function.

3.4 Data-flow Analysis and Data Collection

In this step, represented in Figure 3, we perform the bulk of the analysis and produce a file containing in-depth data about the executed system calls, the internal functions of the samples, their read and written buffers, and their data dependencies, which are examined during the *interactive data exploration* step. During this step, the recording is replayed, collecting and logging *events* and performing taint analysis to find dependencies between them.

An *event* is defined as a series of instructions executed on a given thread in precise time interval. Events can either be: **Syscalls** which cover the activity of kernel-space code from a *sysenter* to a *sysexit* instruction; **Encoding functions** which cover the activity of a known data-transformation function, from their call to return; or **Common functions** which cover the activity of a function taking as input tainted data. Each event is identified by a progressive ID, *label*, an entry point, its start and end times, its process and thread IDs, a call stack, an optional *tag*, and a set of read and written buffers with their associated taint labels, indicating dependencies on previous events.

We call *active events* the events that, at a given time, have started but not yet ended. If the start of an event is detected on a thread where there is already an active event, the new event can either be ignored, or stacked on top of the previous event, on a per-thread stack of active events. When an event terminates, it is removed from the stack.

We define *current event* for a given thread the topmost event in the per-thread event stack. If on a thread there is a *current event* (i.e., the event stack is not empty), all memory accesses happening on that thread on the user-space portion of the address space are attributed to the *current event* and logged.

All reads or writes to the kernel-space portion of the address space are ignored since we are only interested in the data entering or exiting the regions of memory under the control of the user-space malware code we are studying. Events are stacked or ignored according to these rules: (1) If there is no *current event*, any event can be set as such, (2) If the *current event* is a *common function*, any new event can be stacked on top of it, (3) If the *current event* is a *syscall*, no event can be stacked on top of it - i.e., any nested *syscall* is discarded, (4) If the *current event* is an *encoding function*, only *syscalls* can be stacked on top of it.

The collected data takes the form of a list of *events*, each logged together with their input and output buffers and their call stacks. The input buffers can have taint labels attached, indicating the list of previous events that generated or transformed the data they contain. The logged events are either *system calls*, *encoding calls*, or *common functions* and can optionally have a tag indicating that they happened during a call to a high-level API.

In the following paragraphs we detail how *SysTaint* uses taint tracking, how it uses the information optionally provided by the existing sandbox software, and finally how it tracks system calls and their output.

3.4.1 Taint tracking. When a memory access is detected in the user-space portion of the process' address space, on a thread where there is a *current event*, the memory access is attributed to the event and logged. If the memory access is a *read*, the taint labels for the involved memory locations are fetched and logged together with the read data. If the memory access is a *write* and the event is either a *syscall* or an *encoding call*, the memory locations involved are tainted with the ID of the current event. Once assigned, the taint labels are propagated by the taint tracking instrumentation, copying them to a new location when the corresponding data is copied, and mixing them when data having different taint labels is combined – for example as result of an addition. Despite being simple and not requiring any knowledge of the arguments of syscalls or functions, we demonstrate the effectiveness of our approach in the experimental evaluation (see Section 5).

Taint labels are used as follows: if a buffer read by the event *E1* is tainted with the label *E2*, it means that the contents of the buffer are derived from data written by the event *E2*, and therefore the event *E1* depends on *E2*. *E1* and *E2* can be thought as nodes in the data-flow graph, with an edge from *E2* to *E1*.

In order to keep memory usage under control, we assign taint labels only to the outputs of *syscalls* and *encoding call* events, and not to *common functions*, during which we only propagate the existing labels. Despite not applying taint, and thus not appearing as a dependency to later events, the *common functions* can be still effectively queried to understand how the tracked data is transformed and to manually uncover undetected encryption functions.

3.4.2 Interaction with existing sandbox software. *SysTaint* can optionally take advantage of instrumentation, such as the one provided in most sandbox products, that runs inside the virtual machine, along the sample. This instrumentation uses function hooking to monitor the sample's use of known windows API functions, and logs them to a *behavioral log* that the analyst can use to examine an overview of the sample's behavior and select the API calls he wishes to inspect further. Through the integration with *SysTaint*, the analyst can find the *syscall events* corresponding to the selected API calls, examine their arguments, the code of the functions in the call stack, and follow the data-flow of the involved data.

This kind of instrumentation interacts with our tool using *hypercalls*, special instructions intercepted by the hypervisor, to notify *external events* – for example when the sample calls one of the monitored high-level API function. The notified *external event* contains an identifier used to cross-reference the entries in the sandbox's behavioral log from *SysTaint*'s events and vice versa. *SysTaint* treats external event identifiers as *tags*, and maintains a separate tag stack for each thread. When an event (for example a *syscall*) starts on a given thread, *SysTaint* assigns it the topmost tag of the tag stack for that thread, marking it as part of the API call recorded in the sandbox's behavioral log.

3.4.3 Syscall handling and tainting. Syscall events cover the instructions executed in kernel-space between a *sysenter* and a *sysexit* instruction, which respectively move the control to the kernel and back to the user-space code. Syscall events allow logging and taint-labelling all data that enters a process' own virtual memory, and act as the main tap/sink points for taint analysis purposes and as nodes in the resulting data-flow graph. Since *syscalls events* only cover the execution of kernel-space code and we only keep track of user-space memory, we are able to precisely identify and taint-track the program's data involved in the syscall, ignoring unrelated changes to both kernel-space and user-space global data structures. The main downside of working at the syscall level is

that on Windows a syscall by itself gives little semantic information to the analyst, especially considering that syscalls' arguments are most often pointers and opaque identifiers. To understand what is happening at a higher level, the analyst either needs to refer to higher-level functions in the call-stack, or consider the adjacent syscalls in the dependency graph. As an example of this last method, from a `NtReadFile` syscall reading some data of interest it is possible, following the taint label assigned to the file descriptor, to find the `NtOpenFile` syscall, which reads a buffer containing the path of the file to open.

3.5 Interactive Data Exploration

Once *SysTaint* completes the data-collection phase, the analyst can query the collected data interactively to study the malware behaviors, the involved functions, the data processed, and its flow through the execution.

To use the obtained data-flow graph to explore the execution, the analyst should first identify a set of *events*, corresponding to the external interactions identified in the *activities of interest selection* phase. These can be located, among the collected data, in several ways:

- by tag, from a high-level API in the sandbox's behavioral log;
- by searching the events by some known data they processed;
- by filtering the list of recorded events directly, for example finding the encryption call or the `NtDeviceIoControlFile` syscall (which transfer data to and from a socket) that has read the most data;
- by parent function, looking for events having a given address in their call stack.

Once an event E to use as starting point has been found, there are several ways to explore the execution.

- by following the data-flow graph backward, to find the functions producing or transforming the inputs of E;
- by following the data-flow graph forward, finding the functions using the outputs of E;
- by looking for events with the same calling functions in the call stack;
- by looking for *common functions* writing to the addresses corresponding to the input buffers of E.

By moving from an event to another as described, the analyst is, for example, able to locate the event encrypting a buffer starting from the event sending it through the network. Given a buffer, it is possible to find the syscalls that originally produced its contents, as well as the functions writing directly into it.

Inspecting the data-flows can help the analyst to quickly build an understanding of the internal malware functions, and of the behaviors they are involved in. By using *SysTaint* in conjunction with an interactive disassembler, the analyst can directly inspect the relevant malware code and annotate it with its findings. Conversely, from the address of a function in the disassembler, the analyst is able to inspect its calls in a time-traveling fashion, inspecting the read and written buffers, their provenance, and the call-stack information.

4 IMPLEMENTATION DETAILS

SysTaint is implemented as a collection of plugins for the PANDA analysis framework and Python scripts for analyzing the collected data. For simplicity of implementation, we decided to only support Windows7 SP1 on the x86 architecture as guest OS.

The PANDA analysis framework is an open source fork of QEMU offering a platform for full-system (i.e., operating on the entire virtual machine instead of a single process) reverse engineering. It was presented by Dolan-Gavitt et al. in [9]. It offers advanced record-replay features, and a plugin API that makes it easy to perform several kinds of analysis. PANDA plugins can instrument the execution by registering callbacks, called at specific points in QEMU code, for example after a basic block of code has been translated, or before a memory write.

The main steps of the analysis are carried out by the plugins *fnmemlogger*, which collects statistics about the executed functions, that are then used for the cryptographic functions detection, and *systaint*, which performs the actual data collection and taint tracking, producing an output file which can be interactively queried. Other plugins, like *stringsearch2* are instead used during the preliminary analysis, or provide common functionalities.

4.1 Taint tracking implementation

To keep memory usage low and gain additional flexibility, we employed a TCG-based taint-tracking implementation, based on the one used in QTrace[24]. Notably, we excluded from tainting some registers like *ESP* and *EBP* to avoid detecting faux data dependencies between unrelated functions that added or removed from these registers then used them to access the stack's contents. As a tradeoff, our implementation works at byte-granularity, and, due to difficulties in instrumenting the x86 instructions that QEMU implements as C helpers[9], it may propagate taint incorrectly in some edge cases. We tested our taint implementation as described in Appendix A, and worked around these limitations by supporting taint analysis with other methods, as detailed in section 5.

4.2 Integration with Cuckoo Sandbox

We configured *Cuckoo Sandbox* to use PANDA instead of QEMU as hypervisor, and customized *Cuckoo Monitor* – the in-vm instrumentation component of Cuckoo Sandbox which employs function hooking to monitor the execution of the sample and log calls to selected windows APIs – to link its behavioral log with SysTaint's collected data. In particular, we modified *Cuckoo Monitor* so that it sends *hypercalls* to *systaint* before and after it intercepts and logs a call to an interesting Windows API.

4.3 Encryption Functions Heuristics

The *fnmemlogger* plugin is tasked to gather statistics about every called function (tracked through per-thread shadow stacks). These statistics are then used to decide if a given function is a cryptographic one. To keep the analysis time low, only the first 10 calls of any given function are analyzed. *Fnmemlogger* keeps a record for each function call, and updates it after every memory access and executed basic block, keeping track of read and written buffers, counting the times each of the function's basic blocks is executed,

and the number of arithmetic and non-arithmetic instructions executed. When the function call returns, the read and written buffers are summarized by their length, entropy, bytes in the ASCII range, and null bytes, and the per-block execution counters are used to compute the maximum, the total, and the number of distinct blocks executed.

4.4 Considerations on Parallelization

Due to the design of QEMU, both our compute-intensive instrumentation and the guest's code are run in a single thread. To make use of multiple processing cores, the work must be split among several PANDA instances, by either splitting the recording in time-based chunks and analyzing them individually, or running the analysis on the whole recording, but on a subset of the processes. Unfortunately, working on chunks of the recording implies a loss of the internal state, and consequently of callstack and taint information. String searching, by contrast, is essentially stateless and can be easily parallelized.

5 EXPERIMENTAL EVALUATION

5.1 Goals and challenges

The goal of this experimental evaluation is to assess the effectiveness of *SysTaint* to help security researchers in the analysis of the functions involved in the communication and the data exchanged by malware samples. Since using *SysTaint* requires to interactively explore the collected data, we tested it against four real-world malware samples. In particular, we focused on:

- Checking the effectiveness of our heuristics in detecting the cryptographic functions used by the malware process.
- Checking that it is possible to retrieve the unencrypted messages the malware sent and received from the network.
- Checking that the contents of the plaintext messages to be sent through the network are correctly annotated with their provenance.

The main challenge in analyzing real-world malware samples lies in the absence of a ground truth in the form of source code or documentation. This is especially true about the communications with the C&C server, which are rarely described in details. To address the lack of ground truth, we included in our tests a sample of *Zeus*, a *banking Trojan* whose source code is publicly available.

5.2 Experimental Settings

5.2.1 Analysis environment. We executed our experiment on a Windows7 SP1 x86 virtual machine with 512mb of RAM and 20GB of hard disk, with Microsoft Office 2007 installed, disabling Windows Firewall, user account control and Windows Defender. Network and Internet access was provided to the virtual machine, allowing it to communicate with the host and the Internet.

5.2.2 Malware sample selection. We selected four modular trojans known to exchange encrypted binary data in a complex format with their C&C server: *Zeus*, *Citadel*, which we built from source, and *Dridex* and *Emotet*, whose samples we downloaded from VirusTotal [2]. We chose *Dridex* and *Emotet* by filtering a larger set of candidate samples. First, we took a list of malware families known for exchanging complex data with the C&C server from [15], discarding the samples known for employing complex evasion techniques (e.g., *Nymaim*, *Trickbot*). Then, we searched VirusTotal and VxStream sandbox [4] for recent samples of the selected families. Finally, we downloaded the selected samples from VirusTotal in PE/exe x86 format, and ran them in our testing environment, discarding the ones that did not successfully exchange data with any external server in 10 minutes of execution. Only the Dridex and Emotet samples passed this last filtering step. The reason why only two of the samples in our initial dataset communicated with their C&C, even in a 30 minutes time window, can be that they may have either detected the virtual environment (which cannot be hidden completely), employed arbitrarily long timeouts as a measure to evade the analysis, expected to be launched in a specific way (e.g., from a Microsoft Word macro), or had their C&C already taken down.

[2] https://www.virustotal.com

5.3 Experiment 1: Analysis of Zeus and Citadel

Zeus is a banking Trojan whose source code was publicly leaked in 2011, and which became the base for several malware families developed later on.

We built Zeus from source and ran it via Cuckoo sandbox, obtaining a recording covering approximately 10 minutes of execution and 32,811 million instructions. We then gathered, for each process, the code, imported functions, and memory maps at the time of its last instruction.

Looking for network-related API calls in Cuckoo's behavioral log, we inferred that Zeus infected the "explorer.exe" process, and performed its activities from there. We also identified the API call "HttpSendRequestA" sending encrypted data to our server.

Our heuristics detected as likely cryptographic 6 functions that were located in executable and non-file-mapped memory regions, thus likely to contain the injected malicious code. By inspecting these functions with a disassembler, we could identify the RC4 encryption function, and a CRC32 implementation.

We ran the *systaint* plugin, and interactively explored the data it collected by means of a set of Python scripts. In particular, we focused on the syscall handling the largest buffer, among the ones involved in the "HttpSendRequestA" call, identified earlier. We noticed, examining the taint labels on its read buffers, that all the data being sent was processed by a RC4 encryption function. Repeating the process with the buffer in input to RC4, we noticed more encrypted data, with labels identifying the sub-buffers and their origin, but not the encryption function, which we instead found by searching for the last function writing into the output buffer. Comparing this last function to the Zeus source code, we identified it as `BinStorage::_pack`, a complex function which contains an in line call to `visualEncrypt`, an obfuscation function performing the *xor* of each byte with the previous one.

Looking at the plaintext data read by `BinStorage::_pack`, we could distinguish a list of processes, some information about the PC, and a list of HTTP cookies. Among the dependencies of these buffers, we noticed a second invocation of RC4, and a `NtReadFile` call. We could then infer that, before being sent, the data was read from a file (written by another process), decrypted and re-encrypted.

Looking up the `NtReadFile` call in Cuckoo's behavioral log (or equivalently by examining its dependencies), we could find what file the original buffer was read from.

By closely inspecting the dependency graph and the buffers that each syscall reads, it is possible to distinguish the `NtDeviceIoControlFile` syscall sending the header of the HTTP request, having as input the IP address of the host, which in turn is taken from a registry key holding the configuration. In addition, it is possible to distinguish the `NtDeviceIoControlFile` syscall sending the body of the POST request.

It is worth noting that the plaintext buffers and encryption functions can also be easily found if a part of the plaintext can be guessed. In this case, for example, we could have looked for large buffers containing the name of the PC.

The analysis of Citadel was analogous, with only minor differences in the detected encryption functions and identified cleartext data.

5.4 Experiment 2: Analysis of Dridex

The Dridex sample did not produce network activity when analyzed through Cuckoo, but we could observe network activity by starting the sample manually, possibly indicating that the sample detected Cuckoo sandbox.

Since we could not use Cuckoo's behavioral log, we started from the network traffic and using string searching to detect the process that generated it.

The sample made four TLS connections to different IP addresses and did not resolve any DNS name. While the first connection had no response, the second one contained reasonably long and encrypted requests and responses, so we decided to focus on it. We extracted the two payloads sent and the one received, and used string searching to look for memory accesses involving them.

By manually inspecting the matches, we could quickly find the relevant process and some functions in the sample's own code handling the encrypted data verbatim. We noticed that some of the functions found belong to the "bcryptprimitives.dll" library, which contains some known Windows encryption functions. The cryptographic function detection reported, in addition to the known Windows cryptographic APIs, 33 functions in the sample's own code. After running the *systaint*, we searched for the function producing the two sent TLS payload and reading the received one.

The two payloads were encrypted by a function in "bcryptprimitives.dll" and contained, respectively, the header of the HTTP POST request, and a longer encrypted buffer, indicating that Dridex sent an encrypted payload inside of the already encrypted HTTPS request. This last encryption function was not automatically detected, but we could find it as the last function writing into the buffer. In input to this function, we found a password, presumably used for the encryption, and the complete plaintext buffer containing a list of the installed programs, the output of the command "whoami.exe /all" and several other information about the system.

Applying the same procedure for the response TLS payload, we found, to our surprise, that the reply was an HTTP 403 (Forbidden) error message from a Nginx web-server, indicating that our Dridex sample was not able to reach its C&C server.

5.5 Experiment 3: Analysis of Emotet

We were able to record the execution of the Emotet sample via Cuckoo. However, Cuckoo was not able to detect and track all the infected processes. As a result, we had to proceed, as with Dridex, without the information collected by Cuckoo.

The sample made several non-TLS HTTP POST requests, downloading a 1,147 KB payload, probably an updated version of itself, and a smaller 4,692 bytes payload, on which we decided to focus. The sample then attempted to contact several SMTP servers.

Using string searching with chunks of the second payload, we discovered that our implementation could not see the memory accesses writing the received data to user-space buffers, which were therefore not taint tracked. However, the matches indicated that the received data was processed by a function located in non-file-mapped memory area containing the *openssl-1.1.0f* DLL. We added this function to the list of known encryption functions and re-ran the *systaint*, so that it would record and taint its output. We found a second level of encryption/obfuscation, and then, following the taint labels, the plaintext buffer, where we could distinguish a list of email addresses and an email template.

5.6 Experimental Results Discussion

Taint tracking. Our implementation was able to correctly track all the simple data-movement functions and a subset of the more complex cryptographic functions. As expected from the observations in subsection 4.1, it was not able to reliably track data across functions performing complex bitwise operations as AES, and more surprisingly also the much simpler CRC32. Despite these imperfections, our taint tracking implementation proved to be effective in analyzing real-world malware sample, and paired with string searching allowed us to successfully reconstruct the data flows.

Cryptographic function detection. The automated detection of cryptographic functions allowed us to easily deal with at least one level of encryption in all the analyzed samples, speeding up our analysis. However, in all samples, it failed to identify some encryption function in the data-flow path we were interested in, forcing us to find the functions of interest by searching for the *common functions* producing a given sequence of bytes or writing in a given memory area. In the case of Zeus and Citadel, this was due to the "visualEncrypt" obfuscation function being inlined in a larger, non cryptographic function. In Dridex, instead, the undetected function only performed an obfuscation step, but no actual encryption. We could not determine the reason in the case of Emotet as we were not able to inspect the encryption function since, during the analysis, by employing string searching, we identified a wrapper for the encryption function and not the encryption function itself. This may be due to the encryption function not reading the input bytes in sequential order.

Insights on the usage of network communications. In all cases it was possible to navigate the execution and locate the plaintext buffers, whose contents were correctly annotated with their provenance. As expected, not all the contents of the inspected buffers were annotated with taint labels, as not directly derived from tainted buffers. We can classify the data portions without provenance annotations as: (1) constant strings copied from the memory regions where the malware program unpacked itself, (2) field separators

and other constant parts of the protocol, (3) small integers, with a specific meaning, that were not obtained by direct transformation of tainted data.

Even though the analysis of the contents of the fields in the third category is valuable to understand the protocol, to keep the resource requirements acceptable, *SysTaint* limits the use of taint tracking only to the data obtained via syscalls or produced by cryptographic functions.

5.7 Performance

The longest step of the analysis is running the *systaint* plugin, performing the data-flow analysis and data collection, because of the heavyweight taint instrumentation. Table 1 shows the time spent by the *systaint* plugin in the analysis of the samples. The analysis of Dridex is faster in proportion because of the absence of Cuckoo Agent and Cuckoo Monitor from the recording. We believe it is possible to improve the analysis time by employing some simple measures, like disabling taint instrumentation on the processes that are not being tracked.

Memory is a bigger concern, as it depends on the amount of addresses that have been tainted; in our tests, however, it stayed under 8GB.

6 RELATED WORKS

Record-replay. Record and replay techniques are usually employed together with debuggers, allowing the analyst to go forward and backward in the execution trace inspecting the internal states at any point in time, and studying behaviors that are difficult to replicate such as network communications and those carried out by multiple threads. Record-replay solutions are expensive in terms of memory and disk space, and efficient record-replay solutions are relatively recent(*QIRA* [14], *Mozilla RR* [23], *WinDBG* [22], *PANDA* [9], *RAIN* [16]), and still limited to some scenarios, environments or kinds of programs.

Among these, *PANDA* is the only solution that allows recording the execution of a whole virtual machine, ensuring that all the inter-processes and network interactions are captured, and available for later study. Severi et al. [30], showed how efficient record-replay tools such as *PANDA* can be employed to automatically build a dataset of malware traces for later study and reverse engineering.

Cryptographic function detection. Detecting and recognizing cryptographic function is fundamental in the study of malware, and allows extracting the unencrypted data for further analyses. Several works discuss automated techniques to detect encryption functions by means of both static and dynamic analysis.

In [21] Lutz employed dynamic instrumentation and various heuristics (monitoring entropy, known constants, loads and stores, and loops) to locate the decryption functions, using taint analysis to reduce the number of candidate functions.

In ReFormat [35], Wang et al. propose a way of locating the unencrypted buffer by tracing a program's execution while it processed a message, and automatically separating the execution in two phases: decryption and processing by monitoring the ratio of arithmetic operations executed.

In Dispatcher [5], Caballero et al. locate the cryptographic functions by employing a simple heuristic considering the ratio of arithmetic instructions in the body of each function.

Gröbert [13] summarizes various static and dynamic methods, implements and tests the effectiveness and performances of the previous approaches as well as signature-based methods and some newly proposed heuristics.

Li et al. [19] propose a method to detect plaintext buffers in applications that immediately re-encrypt the plaintext buffer after use. Their proposed approach consists in detecting the avalanche effect of encryption functions.

Wang et al. [34] propose an approach to automatically break DRM implementations, allowing the analyst to retrieve the unencrypted media file. They achieve this goal by detecting the decryption functions by means of statistical analysis on the entropy of the written data in order to distinguish encrypted and compressed data.

In *SysTaint* we employ a custom heuristic, similar to [13, 21], to detect both cryptographic and obfuscation functions, based on information on the buffers a given function read or wrote, the function size and instruction composition, the presence of loops and the shape of the call-graph.

Data-flow introspection. Understanding how a process exchanges and transforms data is a key part of the study of its behavior. As such, techniques to extract and study a program's data-flow have been employed in a large variety of works.

Particularly interesting for the scope of this work are some works in the field of automated protocol reverse engineering, which need to find the buffer holding the unencrypted message through the techniques discussed in the previous paragraph, then track its provenance, transformations and usage to infer its format and semantics. Although we do not perform automated protocol reverse engineering, we aim to extract useful and rich data-flow information that the analyst can use to locate interesting buffers and functions and infer the semantics of data. As such, the following works have been a major inspiration for *SysTaint*.

In AutoFormat [20], Lin et al. propose observing the way a program processes a message to automatically identify the fields of the protocol and the hierarchical relation among them. The authors employed taint analysis to track the transformations of each byte in the received buffer, and call stack analysis to determine the context in which each byte was processed.

In Dispatcher [5], Caballero et al. propose making use of dynamic slicing to understand how a buffer is composed. Dynamic tainting was also employed to associate the fields in the deconstructed buffer with selected API calls to infer their semantic.

Memory Forensic. Memory Forensic is widely employed in malware analysis, with frameworks like *Volatility* [11] and *Rekall* [3] including modules to automatically detect signs of infection, for

Table 1: Time spent by *SysTaint* to analyze each sample.

Sample	Instructions	Execution time	Analysis time
Zeus	32,811 M	10 minutes	7 hours
Citadel	12.414 M	4 minutes	2 hours
Dridex	6,853 M	10 minutes	2 hours
Emotet	21,270 M	4 minutes	4 hours

example the presence of function hooks, or known patterns in memory.

These techniques can be automatically employed as part of a sandbox analysis, by applying them, for example, to a memory snapshot taken at the end of the execution. Some commercial sandboxes, like *VxStream Sandbox*[4] also make use of memory analysis to automatically inspect the code being executed.

Teller [33] proposes employing a differential memory analysis approach consisting in configuring a sandbox to trigger a full-system memory dump when certain conditions verifies, then analyzing the changes between the different dumps.

In *SysTaint* we employ memory forensic techniques to perform virtual machine introspection, obtaining the memory maps and contents of the running processes at given points in time, providing the analyst with information on the memory areas and with a quick way to recover the malware code being executed.

7 LIMITATIONS AND FUTURE WORKS

7.1 Limitations

Tracking data not obtained through syscalls. To keep memory and disk usage acceptable, *SysTaint* only tracks the data written by syscalls or by detected cryptographic functions, and not other data that is not directly derived from those. For example, if the malware reads a file, then sets a variable to a constant depending on the contents, *SysTaint* cannot show any dependency between the constant and the contents of the file.

Code employing virtual machines. Our approach assumes that the code of the sample is partitioned in functions that are identifiable by their memory address. This assumption may not hold true for non-native code that is interpreted or executed by a virtual machine, as is the case with higher level languages like Java, but also for programs using complex packing techniques. Additionally, since we rely on the OS's thread identifiers to distinguish concurrent activities and maintain the shadow stacks, mechanisms like coroutines, that multiplex unrelated activities on the same OS thread are also an obstacle.

Evasion. Our approach shares some of the limitations of sandbox analysis. In particular, it requires the sample to manifest its malicious behaviors while it is under observation. To mitigate this issue, our approach can be integrated with techniques such as [17, 18, 25, 31]. Also, malware samples that, as an evasion technique, perform some time-consuming, CPU-intensive, or API-intensive activities can also significantly hamper the analysis, causing both the amount of collected data and analysis time to grow long. For example, as described in [29], *Nymaim* is a trojan employing *Win32 API Hammering* as an evasion technique.

7.2 Future works

Narrowing the semantic gap. It is possible to analyze the output of *SysTaint* to automatically detect and highlight interesting data relationships and behaviors and then allow the analyst to quickly zoom on the relevant code. Moreover, *SysTaint* could automatically infer semantic information about each function from the data it accesses and from the system APIs it uses.

Integration with other software. Allowing the collected information to be accessed from popular reverse engineering environments like *IDA Pro* could be of great help to analysts, who would be able to rapidly check the data any given function read or wrote, complete with provenance information. It could also be possible to leverage the Unicorn Engine [3] to allow the execution and fuzzing of individual functions, using the data that *SysTaint* collected from the function's recorded invocations. Time-traveling debugging could be implemented by adding checkpointing capabilities to *PANDA*, and using the data collected by *SysTaint* as an index into the execution.

8 CONCLUSIONS

In this work, we aimed at building a tool to help the analyst in the malware reverse engineering process, addressing, in particular, the problems arising when debugging and studying malware communicating with external network resources.

We proposed *SysTaint*, a framework implemented on top of PANDA, that collects in-depth data about the execution of the sample, enriched with data-flow and call stack information, so that an analyst can interactively query it to learn useful information about the sample's internal functions and data flow.

We tested *SysTaint* against four real-world malware samples and we proved its effectiveness in extracting useful information about their internal functions, find the unencrypted data being sent through the network and understand its provenance. Nonetheless, our approach is not limited to the study of malware, and can be applied to the reverse engineering of any native software. We released *SysTaint*'s source and we believe it can significantly help the analysts in their reverse-engineering efforts, and that future developments can refine *SysTaint* into a valuable addition to the malware analyst's toolbox.

REFERENCES

[1] 2017. McAfee Labs Threat Report December 2017. https://www.mcafee.com/uk/resources/reports/rp-quarterly-threats-dec-2017.pdf

[2] 2018. Cuckoo Sandbox. https://cuckoosandbox.org/

[3] 2018. Rekall memory forensic framework. http://www.rekall-forensic.com/

[4] 2018. VxStream Sandbox. https://www.payload-security.com/products/vxstream-sandbox

[5] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM.

[6] Andrea Continella, Michele Carminati, Mario Polino, Andrea Lanzi, Stefano Zanero, and Federico Maggi. 2017. Prometheus: Analyzing WebInject-based information stealers. *Journal of Computer Security* (2017).

[7] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: A Self-healing, Ransomware-aware Filesystem. In *Proceedings of the Annual Computer Security Applications Conference* (2016-12). ACM.

[8] Brendan Dolan-Gavitt. 2007. The VAD tree: A process-eye view of physical memory. *Digital Investigation* 4 (2007), 62–64.

[9] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM.

[10] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM.

[11] The Volatility Foundation. 2018. Volatility Framework - Volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility.

[12] Mariano Graziano, Corrado Leita, and Davide Balzarotti. 2012. Towards network containment in malware analysis systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM.

[3] https://github.com/unicorn-engine/unicorn

[13] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated identification of cryptographic primitives in binary programs. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer.
[14] George Hotz. 2016. QIRA. http://qira.me/
[15] Jarosław Jedynak. 2018. Mtracker - our take on malware tracking - CERT Polska. https://www.cert.pl/en/news/single/mtracker-our-take-malware-tracking/
[16] Yang Ji, Sangho Lee, Evan Downing, Weiren Wang, Mattia Fazzini, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2017. Rain: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.
[17] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: efficient malware analysis on bare-metal. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM.
[18] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection.. In *Proc. of USENIX Security*.
[19] X. Li, X. Wang, and W. Chang. 2014. CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution. *IEEE Transactions on Dependable and Secure Computing* 11, 2 (4 2014), 101–114. https://doi.org/10.1109/TDSC.2012.83
[20] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the 15th Symposium on Network and Distribuited System Security (NDSS)*.
[21] Noé Lutz. 2008. Towards revealing attacker's intent by automatically decrypting network traffic. (2008).
[22] Microsoft. 2017. Time Travel Debugging in WinDbg. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview
[23] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association.
[24] Roberto Paleari. 2014. Introducing QTrace, a "zero knowledge" system call tracer. http://roberto.greyhats.it/2014/03/qtrace-part1.html
[25] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*. Springer.
[26] Davide Quarta, Federico Salvioni, Andrea Continella, and Stefano Zanero. 2018. Toward Systematically Exploring Antivirus Engines. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.
[27] Christian Rossow, Christian J Dietrich, Herbert Bos, Lorenzo Cavallaro, Maarten Van Steen, Felix C Freiling, and Norbert Pohlmann. 2011. Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM.
[28] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE.
[29] Joe Security. 2016. Automated Malware Analysis - Nymaim - evading Sandboxes with API hammering. https://www.joesecurity.org/blog/3660886847485093803
[30] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. 2018. Malrec: Compact Full-Trace Malware Recording for Retrospective Deep Analysis. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.
[31] C Spensky, H Hu, and K Leach. 2016. LO-PHI: Low Observable Physical Host Instrumentation. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[32] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. 2009. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM.
[33] Tomer Teller and Adi Hayon. 2014. Enhancing automated malware analysis machines with memory analysis. (2014).
[34] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2013. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services.. In *Proceedings of the USENIX Security Symposium*.
[35] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. 2009. ReFormat: Automatic reverse engineering of encrypted messages. In *Proceedings of the European Symposium on Research in Computer Security*. Springer.

APPENDIX

A CRYPTO FUNCTIONS DETECTION

Due to the complexity of cryptographic function detection and taint tracking, we first evaluated their effectiveness on a test application that makes use of cryptographic APIs from various libraries. For each tested function, we evaluated whether our custom heuristic (*heuristic 1*) and the ratio of arithmetic instructions (*heuristic 2*) can detect it or one of its primitives, and whether our taint tracking implementation detects that the output buffer is derived from the input buffer. As shown in Table 2, our *heuristic 1* detected all the tested cryptographic functions. However, we were not able to obtain the expected results from *heuristic 2* described in [13] and [5].

function	heuristic 1	heuristic 2	taint tracking
wcsncpy	no	no	ok
utf8Encode	no	no	ok
base64Encode	no	no	ok
strlen	no	no	no
CRC32	no	no	no
xor previous byte	ok	no	ok
deflate	no	no	ok
RC4	ok	no	ok
CryptEncrypt AES	primitives	–	no
TinyAES	ok	primitives	ok
OpenSSL AES	ok	no	no
OpenSSL SHA256	ok	ok	ok

Table 2: Results of the taint-tracking and encryption detection tests