FORMAL REPRESENTATION AND COMPARATIVE ANALYSIS OF SOFTWARE RUNTIME PROCESS

O. Baranovskiy\(^1\), R. Tkalenko\(^2\)

\(^1\)National Technical University of Ukraine “Kiev Polytechnic Institute”
37, Peremohy Avenue, Kyiv 03056 Ukraine
o.baranovskyi@kpi.ua

\(^2\)Institute of Physics and Technology, Ukraine
r@tkroman.com

Abstract: Using a software tracing frameworks we obtain sequences of system calls produced during the execution of the pair of programs. We then transform the sequences of system calls into the symbolic sequences and apply a set of string metrics to them. We experimentally compare metrics on the task of differentiating the sources of sequences. We investigate various metrics including edit-distance metrics and hybrid methods. Overall, the best-performing metric is a cosine distance, resulting in order-of-magnitude different values for input pairs of different degrees of similarity.

Keywords: software, program verification, formal methods, runtime process, comparative analysis.

Introduction

Traditional types of malicious software, which are viruses, trojan horses, backdoors, rootkits etc., are often combined with a view to hide their presence and activity from anti-viruses. The purpose of the malware activity hiding is also often achieved by the means of embedding malicious code into the non-malicious software. The task of distinguishing such a modification from an ordinary copy of the program is one of the most challenging problems in this field (Horwitz, 1990).

From the analyst’s point of view the following key points are worth noting with respect to the typical aims and/or behavioral features:

- data manipulation and integrity violation (modification, overwriting, removal of files);
- uncontrollable multiplication of instances of the specific file;
- atypical resource usage patterns: temporary ‘spikes’ of the network activity, seemingly causeless high CPU usage, abnormal disk drive activity etc.

Abnormal behavior can also be caused by unintentional errors (bugs in the code). Such deviations are even harder to detect, since the most common unintentional errors don’t usually reflect directly on the behavioral patterns, but rather on how the program reacts to the state of the environment it’s being running in.

An illustrative example to this is implement of backdoor as a module of a regular application. During the normal execution phase no extrinsic features appear and the process behavior is considered to be normal. However, a combination of conditions can trigger a backdoor module execution and an irregular patterns can be observed on a various levels including unusual system calls, data access patterns etc.

Distinguishing these abnormal periods is not an actual goal, but is rather a tool to be used to point out malicious software. In this article we propose an approach based on a comparison of data gathered from reference software to the data collected from an arbitrary source.

This approach is expected to prove itself useful even if a source code of a program is present, since it’s presence doesn’t guarantee that a complete and thorough analysis was conducted. In case a source isn’t available, such an approach appears to be a more efficient and cheap (in terms of time and resources) alternative to the procedure of the decompilation of the software piece in question.

We propose to use a sequence of system calls made by a program during its runtime as a source of comprehensive information about the actions performed. Various evidences of the potency of methods based on the system calls sequences study were reported (Kruegel et al., 2003; Cabrera, Lewis and Mehra, 2001; Forrest, Warrender and Pearlmutter, 1999). Applications include intrusion detection, threat analysis and various predictive algorithms. There are at least two approaches: 1) analyzing just a sequence of calls as a
whole (Hofmeyer, Forrest and Somayaji, 1998) and 2) analyzing arguments to the calls along with the calls’ positions in the sequence (Maggi, Matteucci and Zanero, 2009).

In this work we concentrate on a slightly more specific question: comparison between two or more program behaviors with the aim to determine differences between them based on various marks and patterns typical to that question. Our goal is a finding a function that, given a pair of sequences, outputs a value that corresponds to a degree of difference between them. The results produced by the function should be representative to analysts could easily tag them in terms of “highly similar”, “highly different”, “slightly different” etc., independent of the length of the sequences and other factors. The target function should be the core element of the framework that will allow to obtain sequences and operate them in any way according to carry out the experiment. We define the following requirements to the solution:

- to be un-invasive: the fact of the observation doesn’t affect the process being observed itself;
- to provide an environment for running analysis tools that has a high degree of resemblance to the target environment, but does not allow any personal data threats (a clean virtualized environment);
- to guarantee results to be deterministic and reproducible. This requirement is about analyzer’s insensitivity to some shifts or permutations in a program execution order, which can be an issue that can arise in a multi-threaded environment. This is also addressing a problem of analyzing data in an off-line mode and various sizes of data samples;
- to guarantee results to be independent from the level of difference between two data samples. This means that the value of difference indicator should be a well-defined function of input data, and also makes it possible to provide data from programs with a completely different behavior in order to obtain values specific to the concrete differences, e.g. comparing data from a program that uses a network and the one that doesn’t should yield results that are correlated with the network usage;
- to be immune to attempts of hiding the actual data from the observation. ‘Immunity’ means that attempts of hiding the facts of malicious actions should be perceived as the rest of the data. In such a way these attempts will be reported in the same way as the other, thus guaranteeing the insensitivity to artificial interference.

Method

First of all, it’s worth noting that a system call sequence provides comprehensive information about the software activities and allows not only a runtime analysis, but even when only a collection of log files, which can be useful for analysis, data mining and machine learning purposes. This approach satisfies the requirement of immunity to external interference since most of the attempts to cover the suspicious activities will be recorded as another series of system calls, which, in turn, will show up in a resulting metric value. A theory behind the set of analysis tools to be used during the next stage comes mainly from the field of frequent sequence mining and general set theory.

We obtain a symbolic mapping from the raw data to the set of all natural numbers by assigning each unique call an available symbol. Having an input sequence of system calls

\[ \text{calls} = (c_i), \quad i = 1, N, \quad c_i \in S \]

where \( S \) is a set of all calls, and a function (one may see it as a global registry)

\[ f : S \rightarrow N \]

we derive a function

\[ f(c) = \mu_i cS \in [0,1] \]

This function formalizes input representation, allows to state a problem in terms of obtained abstractions and to apply a number of algorithms dedicated to sequence analysis.

The problem of distinguishing pairwise-irrelevant patterns considered in this article is as follows:

**Having a pair of sequences** \( S_1, S_2 \in S \), **find a function** \( \mu : S \times S \rightarrow [0,1] \) (or \( \mu : S \times S \rightarrow R \)):

- if \( S_1 \) and \( S_2 \) are obtained from programs with similar behavior, \( \mu(S_1, S_2) \rightarrow 0 \),
• if $S_1$ and $S_2$ are obtained from programs with highly different behavior, $\mu(S_1, S_2) \rightarrow 1$.

• sequences retrieved from variations of the same program produce values in range $[0, 1]$.

We represent described sequences with a special symbol ‘-’, which denotes an absence of the symbol (situations in which a need for such a symbol may arise will be described later). Then we focus our research on applicability of string metrics to the problem.

A function $\mu$ described before is called a metric in this article, which is a commonly adopted term for a string distance measurement functions, although some properties of the metric might not hold for a set of examined functions.

A metric $\mu$ is normalized if

$$\mu(S_1, S_2) \in [0,1].$$

and $\mu$ is not normalized if

$$\mu(S_1, S_2) \in [0, \infty].$$

The following string metrics are considered in this article:

• Levenshtein / Damerau-Levenshtein
• Jaro / Jaro-Winkler
• Jaccard
• longest common subsequence
• q-gram
• cosine
• Monge-Elkan

These functions are well known in the field of general string manipulation tasks (Cohen, Pavikumar and Fienberg, 2003).

There is a certain type of interference that needs to be taken into account while considering a metric applicability. Representing an execution progress as a sequence,

$$P = p_1p_2...p_m$$

a concurrent execution of the program subroutines during separate program runs can result in the same summary effect $E$, but sequences themselves will differ:

$$P' = p_1'p_2'...p_m'$$
$$\mu(P, P') = p_1''p_2''...p_m''$$

$$E(P') = E(P'') = E(P) = E$$

In this case it’s important for the metric to be resistant to permutations of the input data, while remaining highly sensible to semantic differences in it.

Metrics description

Levenshtein distance characterizes difference between the $S_1$ and $S_2$ sequences in terms of a number of edits necessary to transform $S_1$ into $S_2$. Any single-symbol insertion, deletion or substitution is considered to be an edit. Application of this metric is effective in case the amount of positions containing different symbols is insignificant relative to the overall length of the sequence. Lower values of the metric correspond to a high degree of similarity between program behaviors, but not the other way around, since this metric is susceptible to permutation interference, hence can only be used in combination with other metrics.

Jaro-Winkler distance is not a metric in the strict sense, since the ‘triangle inequality’ is not maintained. It can prove useful in the task of calculating characteristics based on the amount and order of symbols common to both sequences. The drawback of this function is that its efficiency is constrained to sequences of relatively small length. We’ll introduce a composite metric based on Jaro-Winkler distance, that will overcome this flaw, further in this article. This metric will be used as a diagnosis tool for identifying similarities between segments of sequences when relative shifts are insignificant, which can be applied in cases of multi-threaded applications.
**Q-gram** distance is based on the idea of collecting all the q-grams that comprise the sequence and computing statistics based on the data. Q-partition can be represented as a vector in a space $S_q$, where $S$ is the set of all symbols. For example, having alphabet \{0, 1, 2, 3\} and sequences $S_1 = 102030$, $S_2 = 0021301$ 2-gram (more commonly referred to as bigram) vectors are as follows:

- $S_1^2 = 00\ 01\ 02\ 03\ 10\ 11\ 12\ 13\ 20\ 21\ 22\ 23\ 30\ 31\ 32\ 33$
- $V(S_1, 2) = 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0$
- $V(S_2, 2) = 1\ 1\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0$

Q-gram distance is calculated as a traditional distance between two $S_q$-dimensional vectors. The value of each coordinate is a corresponding q-gram frequency, thus a small distance shows a high degree of similarity between the sequences.

In most traditional applications the fact of information loss (about relative positions of q-grams in sequences) is considered to be a drawback, but for the purpose of this article precisely this feature allows for the resistance to the ‘permutation-induced interference’.

Primary problem that arises in this method’s applications is the uncertainty of the parameter $q$. The task of finding optimal value of $q$ requires either full-space search (i.e. evaluating the algorithm with $q = 1, 2, ..., |S|$) or a certain intervention on the part of the analyst (preliminary assessment of the possible values of $q$ based on the obvious patterns).

This metric is potentially useful for the analysis of the multi-threaded applications and self-mutating viruses that modify the order of execution of sub-modules to avoid detection.

**Jaccard distance** is closely related to the q-gram distance, since its formulated as the ratio of the number of q-grams present in both sequences to the total number of q-grams.\(^1\) In case of an adequately determined value of $q$ this metric is useful for the characterization of the relative degree of similarity in terms of common blocks. Resistance to the permutation-induced interference is preserved. Since this metric is normalized (in sense discussed above), an analyst can use it to compare values retrieved from pairs of sequences of different lengths. An example of why this property is so important is as follows: value of the metric for two absolutely different sequences of length 5 will be less than the value obtained from two sequences of length 500 differing only in 10 positions, but in terms of relative similarity second pair of sequences is differing on just 2% of positions, while the first one has no common elements at all.

**Longest common subsequence distance** is based on a number of symbols that should be removed from $S_1$ and $S_2$ to achieve the equality between them: $S_1 = S_2$. This metric is not stable with respect to permutations, but is useful during the analysis of sequences obtained from different programs. For example, if the reference program doesn’t use network, but an infected version exists that does use the network, and isn’t different from the reference one otherwise, the LCS metric is able to indicate such a feature.

**Cosine distance** is based on the q-gram distance as well. While the canonical q-gram metric is a distance between two vectors, cosine metric is defined as a cosine of the angle between those vectors (hence it’s normalized).\(^1\) This is the most sensitive distance function.

\(^1\)All q-gram-based metrics (more precisely, all functions of vectors in spaces of q-grams) are referred to as distances for the continuity’s sake, but actually correspond to similarity. Distances can be calculated as follows: distance$(S_1, S_2) = 1 - \text{similarity}(S_1, S_2)$

Experimental values correspond to distances as well.

**Monge-Elkan distance** is a hybrid distance function that uses an internal (secondary) metric during the first phase of a computation. It is defined as some combination of maximal distances between blocks of fixed length. The metric is useful for the analysis of sequences of arbitrary length with application of algorithms otherwise inapplicable (e.g., Jaro/Jaro-Winkler metrics).

### Implementation

An open-source toolkit was built for the sequence obtainment and analysis. The core idea of the toolkit is using the low-level tracing framework for the first phase. Any framework that is inspired by the Unix program `truss` (dtruss on Mac OS, strace on Linux, ktrace on *BSD etc.) can be used. Within the confines of this article dtruss (in combination with Mac OS X) in used. The sequence obtained is further passed into a pipeline consisting of the following elements:

1. Excessive information removal (call’s parameters, memory addresses etc.).
2. Call registration (assigning a unique symbol to the call in case one wasn’t previously assigned). Might also involve additional operations to provide more information later.

3. Mapping call sequence into a symbolic sequence.

There is an option to apply a sequence alignment algorithm to have both sequences of equal length and a higher degree of similarity. The next step is actual application of the algorithm.

The toolkit provides a unified API that allows operating on sequences without application of any algorithm as well. As an example of that we provide an instrument for visual inspection of the sequences, which can be used to quickly assess the similarity between the two programs’ behavior and the nature of it.

Experiments

Three typical situations are considered as the models for the experiments:

1. Repeated launches of the same program.
2. Launches of a single program with varying parameters.
3. Launches of different programs.

To demonstrate the practical applicability of the described conceptions, real programs have been selected as input generators. To verify the algorithms for the first type we have chosen `ls` and `ping` (in pairs `(ls, ls)` and `(ping, ping)`), for the second type - `ls` and it’s variants obtained by passing additional parameters and for the third type – `ls` and `ping` have been selected to run against each other. Each run was repeated 100 times and results were averaged afterwards.

Results and Discussion

Results obtained from experiments carried out according to described schemes are shown in figures 1 and 2. Normalized and not normalized metrics are separated to keep adequate scales.

**Fig. 1.** Metric values as functions of program pairs for not normalized metrics

**Fig. 2.** Metric values as functions of program pairs for normalized metrics
Results show that pairs consisting of the same program result in the smallest values of differences, pairs formed by a slight variations of the same program (when additional arguments are passed etc.) result in higher values, and finally, different programs show the highest values. As a rule, thresholds by which the differentiation is performed are empirically specialized for the concrete program. Certain correlation with the length of the input is present as well, however, for normalized metrics this is not an issue. Some ideas on how to get rid of this dependence will be proposed in the next chapter.

Another important property of the metric that should be taken into consideration is relation of its values for different pairs. The most expressive metric from this point of view is the cosine distance, which gives values of different orders for different combinations: $10^{-1}$ for very similar programs, $10^{-2}$ for variations and $10^{-3}$ for distinct pairs. Jaccard and Jaro-Winkler metrics also demonstrate this sort of behavior, but in a less manifested manner.

The visual module presents another important aspect of the runtime behavior - an encoded sequence of calls (a representation the the same sequence that is used as a comparator input). Fig. 3 shows an example of the program interface for the type 3 situation (according to the classification introduced previously) - launch of different programs.

![Fig. 3. The execution sequences from the `ls` and `ping -c16 google.com`. The program provides means for the call information retrieval as well.](image)

As the image demonstrates, both sequences have common parts. These parts are specific not to a single program, but rather to the platform (environment) the program is being run in: the calls present in the common sections are mainly `mmap`, `fcntl`, `stat`, `fopen`/`fclose` etc. - the ones that are used to initialize the correct program startup and shutdown. This is important since it allows to optimize sequences for some particular types of analysis before feeding them as an input, thus eliminating possible interference with the sections of the program that actually bear information.

The comparison of these sections is more interesting from the point of view of this article, since visual assessment of the program output allows seeing distinctive patterns that are specific to each program. An inspection of calls’ metadata shows that the ping output contains a network call section, while the ls sequence doesn’t contain such a pattern. Without the prior knowledge of the sequences’ source this is very useful information that will allow an analyst to judge the behavior of the program based on just the necessary minimum of the data.

It could be argued that examined functions demonstrate a uniform behavior, i.e. they react to the differences in inputs in the same way to some extent. However, another factor that we should pay attention to is the sensitivity of the metric. The cosine distance has the best performance from this point of view, and all the other members of the q-gram family demonstrate a significantly higher degree of sensitivity. The value of $q$ in each concrete case is an important parameter since we analyze not every single system call by itself, but rather patterns that are common to the concrete program, and the closer $q$ is to the length of the typical pattern, the more appropriate results will be achieved.

Since the experimental data was collected from real-world programs, it can be safely stated that the results obtained during the experiment are closely related to those that will arise during the actual application of the ideas shown in the article. However, it should be noted that the run time of the subject programs is significantly less than the run time of a typical server application, hence real implementations will require greater amount of resources or a more optimized analyzer implementation.

We also left out an alignment stage, which can be used to level off sequences by their common parts, which, on one hand, will increase scores that are based on relative positions of segments of the sequences, and on the other hand (on account of gaps introduced during the alignment) will significantly lower scores based on both relative position and q-gram model.
Another limitation of the proposed approach is introduced during the phase of the calls’ arguments striping. We are planning on focusing the next stage of our research on the algorithm that will allow for the differential analysis of the sequential patterns with respect to the actual parameters.

References


