An Efficient Common Substrings Algorithm for On-the-Fly Behavior-Based Malware Detection and Analysis

Jaime C. Acosta, Brenda G. Medina, Humberto Mendoza
U.S. Army Research Laboratory
White Sands Missile Range, NM 88002–5513

Abstract—It is well known that malware (worms, botnets, etc...) thrive on communication systems. The process of detecting and analyzing malware is very latent and not well-suited for real-time application, which is critical especially for propagating malware. For this reason, recent methods identify similarities among malware dynamic trace logs to extract malicious behavior snippets. These snippets can then be tagged by a human analyst and be used to identify malware on-the-fly. A major problem with these methods is that they require large processing resources. This is especially due to the large amount of malware released each year (upwards of 17 million new instances in 2011). In this paper, we present an efficient algorithm for identifying common substrings in dynamic trace events of malware collections. The algorithm finds common substrings between malware pairs in theoretical linear time by using parallel processing. The algorithm is implemented in the CUDA and results show a performance increase of up to 8 times compared to previous implementations.

I. INTRODUCTION

The impact of malware on communication systems can be devastating if not contained early during an attack. The current process of detecting and analyzing malware is very latent and not well-suited for real-time application, which is critical especially for propagating malware. This process consists of capturing, analyzing, and finally recording the malicious binary to aid in future analyses (see Figure 1).

There are several ways to capture suspect binaries. An anti-virus scanner may capture a known binary by using signature matching. In this case the hash value of an executable matches the hash value for a known malware instance and no further analysis of the binary is required. Alternatively, a honeypot (a system intentionally left vulnerable in order to trap malware) is used [1], [2]. A third way is to identify executables that behave out-of-the-ordinary.

After a binary is captured it is analyzed using either static or dynamic analysis methods. Static methods use reverse engineering tools, e.g., [3], to analyze the binary outside of runtime. Dynamic analysis [4] involves executing the binary in a controlled sandbox environment and logging all events during invocation including registry, file system, and socket API calls. While static analysis enables full path traversal (meaning that all possible paths of a program’s execution can be traced), it is extremely rigorous, and quickly falls victim to state-space explosion, especially with novel obfuscation techniques [5]. For this reason, recent work uses dynamic analysis, which has succeeded for behavioral similarity analysis [6].

After analysis, if the binary is malicious, a hash signature is generated (which enables quick anti-virus scanner lookups). Lastly, the behavior is recorded to aid with future analyses.

For known malware, anti-virus scanners are very efficient because the above mentioned analysis and recording steps can be omitted (they have already been done previously). However, these scanners are not capable of identifying even slightly modified and recycled malware. Each year a large number of malware is released; in 2011, there were more than 17 million new instances recorded [7]. Therefore, the analysis and recording steps are bottlenecks in the process. A novel approach that alleviates this problem uses behaviors (not signatures) to identify malware [6]. Implications of this approach include the ability to identify new malware in real-time based on known malicious behaviors (as observed in execution logs). This method has shown success, however, it requires large processing resources for identifying these malicious behaviors (also known as common substrings) among malware. As the number of malware released each year grows, a more efficient way for common malicious behavior identification is needed.

In this paper, we make the following contributions:

1) We provide a description and implementation of the common substrings algorithm written in the CUDA.

2) We show that the common substrings algorithm runtime outperforms previous implementations of the algorithm by up to eight times.

The rest of the paper includes a brief review of research that led up to this work. Next we describe four implementations
of the common substring algorithm and an experimental procedure. We report our results and conclude with future work.

II. MALWARE ANALYSIS REVIEW

With both static and dynamic analysis methods, a current trend is to use machine learning for clustering similar malware. Clustering methods are useful because they generalize large sets of malware into categories with limited need for manual human intervention. Jang and Brumley [8] perform static analysis by identifying areas of code reuse by clustering malware binaries. His clustering method uses bloom filters, which identify similarity of malware instances by applying hashing techniques to fixed size chunks of the malware executable code.

On the other hand, Bayer et al. [9] use machine learning to identify similarities in malware instances by comparing their dynamic event traces, which include system calls, their dependencies, and network behavior. Next, the malware instances are clustered based on their dynamic behavior. A limitation of this approach is that the algorithm is trained with a fixed set of malware. It does not allow retraining with additional malware samples during the clustering phase. Rieck extends this with his Malheur [10] system by establishing an iterative mechanism that consists of clustering and then classifying new instances into existing clusters. In his work, similarity is determined by the presence of shared fixed-length instruction sequences. In addition, Rieck also uses a dynamic trace representation format called MIST [11] that allows prioritizing event parameters (e.g., an open-file system call may have the file name, file type, and the file path as parameters). This is meant to allow more efficient processing for machine learning algorithms by reducing the input file size by leaving out less-critical parameters. MIST also provides a common file format to which many of the available sandbox output can be converted.

After the instances are clustered, an analyst may have to conduct deeper investigation, such as exact differences and similarities in the binaries. It may be the case that malware in different clusters share common behaviors. This results in redundant analysis by a human analyst. Another issue is that instances in a cluster are not exactly the same.

Some techniques that use semantic structure for finding similarity are in code-clone detection research. These techniques have been used to identify redundancy to reduce program size or to identify plagiarism in legitimate software [12]. The problem with using these techniques for identifying similarity and differences in malware is that the source code of malware is not available. Some attempts have been made to analyze the sequences of instructions of disassembled binaries to determine whether they are malicious. One method compared the disassembled code against behavior templates that are known to exist in malware. These templates are able to capture malicious behavior, even if the malware has small variation [13]. Another method [14] uses the Intelligent Malware Detection System (IMDS), to identify malware instances by checking if certain sequences of Application Programming Interface (API) calls exist in a binary Portable Exchange (PE) file. A limitation of both of these examples is that they assume the binary file is not packed and is not virtualized.

To alleviate the former issues, in more recent work, the common substrings algorithm is used to identify common runtime log sequences of variable length among a set of malware. Results showed that the commonalities are not restricted to small sequences; many large sequences are shared among the malware instances, which mean that there are in fact many shared behaviors present that could be identified and used to identify malware based on behavior. An issue with this approach is the long runtime of the common substrings algorithm.

In this paper we provide an efficient implementation of the common substrings algorithm that takes advantage of off-the-shelf graphics processors.

III. COMMON SUBSTRINGS

The goal of the common substrings algorithm is to find and keep runtime log event sequences that are shared among malware collections. These shared sequences can then be tagged with semantic labels by human analysts in order to enable on-the-fly malicious behavior detection.

Figure 2 shows three versions of the common substrings algorithm. The first (Figure 2 (a)) is the original sequential version that uses the dynamic programming scheme in order to reduce redundant processing during common substring identification. Although redundant processing is reduced, the sequential nature of this version still requires long processing times for large inputs. In general, the algorithm works by iterating over each set of characters from two input strings. A matrix is generated with each string on an axes. During processing, if a match is encountered, the value in the top-left position in the matrix is prepended to the current position value. Eventually, the matrix will be filled with all common substrings among the two input strings (these are circled in the Figure). This version of the algorithm is described in full detail in [6].

The second and third versions of the algorithm (Figure 2 (b) and (c) respectively) exploit the parallelism of the problem. As with the sequential version, a matrix is generated. Unlike the sequential version, each diagonal in the matrix (StartDiag) is treated as an input that can be independently run on threads resulting in theoretical $O(\min(n,m))$ processing time when assuming infinite processors ($m$ and $n$ are the sizes of the input strings). The difference between the second and third version of the algorithm is that in version 3, the input matrix is shifted to enable more evenly distributed loads among threads. More specifically, instead of processing each diagonal in the matrix separately, shorter diagonals are combined with longer diagonals.

We have implemented versions 2 and 3 of the common substrings algorithm in order to compare performance with the original version (which was implemented in the Java programming language). We implemented version 2 first using Java and second using the CUDA. We implemented version 3
of the algorithm using the CUDA. In the rest of this section, we describe the implementations of the three algorithms.

A. Sequential

We implemented the sequential algorithm as described in [6]. Some benefits of the previously implemented original version include the ability to run on any machine that can run the Java runtime environment. In addition, the original was shown to run well on small inputs and used a very small memory footprint during processing: $O(\max(m, n))$. As previously mentioned the issues with this implementation lie in the long processing time, which is $O(mxn)$.

B. Parallel

Algorithm 1 contains the pseudo code of the parallel version of the algorithm. The algorithm starts by separating the input into a set of individual diagonals (lines 2–4). Next, each thread calculates a starting point (lines 7–13) and then two tracking variables that contain current common substring position and length are initialized. Each character set in the diagonal are compared. If the characters match, then the two tracking variables are updated (lines 17-22). If there is no match, then any currently tracked substring is added to a result array and finally the tracking variables are reset (lines 23–30). This process is repeated for all items in the diagonal.

1) CPU Thread Implementation: We implemented the parallel algorithm using Java threads. As with the original implementation, the algorithm can run on any machine with a Java runtime and uses a small memory footprint for processing. Informal observations indicated that this implementation ran well with small inputs, but still did not perform well with large inputs.

2) CUDA Implementation: For the CUDA implementation, Java was used as the host code and the JCuda bindings [15] were used for the graphics device interface code. Informal observations indicated that this implementation ran well with small and large inputs.

![Fig. 2. Common substrings algorithms including the sequential (a), parallel (b), and the input shift (c).](image)

Algorithm 1 Parallel Common Substrings

1: procedure PARALLEL(strA, strB)
2: for $i = 0$ to $totalStartDiags$ do
3:   PROCESS_DIAGONAL($i$)
4: end for
5: end procedure

6: procedure PROCESS_DIAGONAL($startID$)
7: if $startID < \text{len}(strA)$ then
8:   $i \leftarrow startID$
9:   $j \leftarrow 0$
10: else
11:   $i \leftarrow 0$
12:   $j \leftarrow startID - \text{len}(strA) + 1$
13: end if
14: $\text{subStrLen} \leftarrow 0$
15: $\text{subStrStart} \leftarrow -1$
16: while $i < \text{len}(strA)$ and $j < \text{len}(strB)$ do
17:   if $strA[i] = strB[j]$ then
18:     if $\text{subStrStart} = -1$ then
19:       $\text{subStrStart} \leftarrow i$
20:     end if
21:     $\text{subStrLen} \leftarrow \text{subStrLen} + 1$
22:   end if
23:   if $strA[i] \neq strB[j]$ or
24:     $i = \text{len}(strA) - 1$ or
25:     $j = \text{len}(strB) - 1$ then
26:     if $\text{subStrStart} \neq -1$ then
27:       addResult($\text{subStrLen}, \text{subStrStart}$)
28:       $\text{subStrStart} \leftarrow -1$
29:       $\text{subStrLen} \leftarrow -1$
30:     end if
31:   end if
32: end while
33: end procedure
include low cost per core and low power usage profiles of graphics processors. A drawback with this implementation is the fact that data (results, input strings, etc.) must be transferred to and from the GPU device. This may limit the number of diagonals that can be processed at one time and may cause additional latencies dependent on specific graphics processors. In our case, we tested with inputs that resembled a real collection of malware traces obtained from [16] and this did not present a problem. Another drawback of this approach is the need for a graphics processor.

C. Input Shift

Algorithm 2 Input Shift Common Substrings

1: procedure PARALLEL(strA, strB)
2:    if len(strA) ≠ len(strB) then
3:        StrA ← longer of the two input strings
4:        StrB ← shorter of the two input strings
5:    end if
6:    for i = 0 to len(strA) do
7:        PROCESS_SHIFTED_DIAG(i)
8:    end for
9: end procedure

10: procedure PROCESS_SHIFTED_DIAG(ID)
11:    PROCESS_DIAGONAL(ID)
12:    secondaryID ← len(strA) + len(strB) − 1 − ID
13:    if secondaryID < len(strA) + len(strB) − 2 then
14:        PROCESS_DIAGONAL(secondaryID)
15:    end if
16: end procedure

The input shift algorithm allows threads to process up to two diagonals in order to more evenly distribute execution load. If the two input strings are of unequal lengths, then StrA, which is along the y axis (see Figure 2) is assigned the longer and StrB is assigned the shorter. In lines 6–8 a thread is started for each diagonal on the y axis, unlike Algorithm 1 that invokes a thread for each diagonal. All threads execute the PROCESS_DIAGONAL method from Algorithm 1 twice, except in the cases where the thread is on the main diagonal and when strings are unequal in size and there are no secondary diagonals (lines 11-15).

1) CUDA Implementation: The implementation is again written using Java and JCuda (as with the Algorithm 1 implementation). Informal observations indicated that this implementation did well with small and large input sizes. The shifting method allows larger inputs to be processed and also facilitates partitioning extremely large inputs across multiple CUDA executions.

IV. PERFORMANCE EVALUATION

We conducted an experiment to compare performance among the four implementations of the common substring algorithm. The host computer was a Dell Precision M4600 with the following specifications:

- Intel Core i5 2520M processor
- 8 GB DDR3 SDRAM at 1600 mhz
- NVIDIA Quadro 1000M

The graphics processor used has the following specifications:
- 96 CUDA cores
- 700 mhz core clock speed
- 2GB GDDR3 dedicated memory

A total of 100 configurations were executed. Each configuration contained two input strings; both equal in size. Sizes ranged from 500 to 50000 characters per string at 500 character intervals, i.e. configuration 1 contained sizes 500x500, configuration 2 contained sizes 1000x1000, etc. These sizes are similar to those observed from a malware dataset [16]. For each configuration, two cases were executed. In the first case both input strings were identical, except for every twenty-first character; resulting in substrings of length 20. In the second case, the two input strings had no matching characters. This was done in order to distinguish character-comparison performance and memory performance. Each configuration was run 20 times (10 for each case) and average times were calculated for overall performance, comparison processing performance and memory performance. Overall performance is based on the times taken in case 1 (where matches occur). Comparisons processing performance is based on the times taken in case 2 (where no matches occur). Finally, memory performance is based on the differences in case 1 and case 2. Also included in memory performance is the time taken to transfer results to and from the graphics device.

In all cases, the Java executables were run with the hotspot flag enabled and 8GB of runtime memory allocated at startup.

V. RESULTS

The following figures present latencies associated with the four implementations of the common substrings algorithm: sequential (the original implementation), threaded (the parallel algorithm implemented using Java-only), CUDA (the parallel algorithm using the CUDA), and finally the CUDA with shift (the input shift algorithm implemented with the CUDA). The x axis values (comparisons) are calculated as the product of the input sizes, e.g. using two strings of size 500 yields 500x500 = 250000 comparisons.

Figure 3 contains the overall performance results. In general, it can be observed that both CUDA implementations perform roughly eight times faster than the Java-only implementations for large inputs. The CUDA implementation of the input shift algorithm performs slightly faster than the non-shift version. This is probably due to the low CUDA core to thread invocation ratio. The CUDA implementations also exhibit a seemingly linear growth as compared to the Java-only implementations.

When comparing only the latencies associated with comparisons, the CUDA implementations are up to twelve times faster than the Java implementations for large inputs. The Sequential and threaded versions have very similar performance, but as the number of comparisons grows to very large values,
the threaded approach surpasses the sequential. Both CUDA versions have almost identical performance, meaning that the performance improvements with the input shift algorithm are associated with the memory performance; this can be observed further in Figure 5.

An interesting observation is that the sequential and threaded algorithms outperform both CUDA implementation memory performances for small inputs. This makes sense as there is overhead associated with memory exchange between the host and graphics device. For larger inputs the CUDA implementations are faster with the input shift algorithm performing best.

VI. CONCLUSIONS AND FUTURE WORK

We have introduced three novel implementations of the common substrings algorithm. We showed that using the CUDA greatly increases overall processing time by up to 8 times compared to the previous implementation for large inputs.

Future work will look at labeling common substrings among malware and then using these labels along with machine learning to automatically determine previously unseen malicious behavior in real time. We are also developing a graphical interface to facilitate the labeling process. We will also look at using sequence alignment methods (which also takes into account non-contiguous common substrings).

ACKNOWLEDGMENTS

We would like to thank Victor Mena and Mohammed Alawneh for their helpful suggestions during this work.

REFERENCES