Parallelization of Machine Learning Applied to Call Graphs of Binaries for Malware Detection

Robert Searles, Lifan Xu, William Killian, Tristan Vanderbruggen,
Teague Forren, John Howe, Zachary Pearson, Corey Shannon, Joshua Simmons, John Cavazos
University of Delaware
Department of Computer and Information Sciences
Newark, DE
{rsearles, xulifan, killian, tristan}@udel.edu
{tforren, jhowe, zpearson, cshan, jsimmons, cavazos}@udel.edu

***Abstract*—Malicious applications have become increasingly numerous. This demands adaptive, learning-based techniques for constructing malware detection engines, instead of the traditional manual-based strategies. Prior work in learning-based malware detection engines primarily focuses on dynamic trace analysis and byte-level n-grams. Our approach in this paper differs in that we use compiler intermediate representations, i.e., the call-graph representation of binaries. Using graph-based program representations for learning provides structure of the program, which can be used to learn more advanced patterns.**

We use the Shortest Path Graph Kernel (SPGK) to identify similarities between call graphs extracted from binaries. The output similarity matrix is fed into a Support Vector Machine (SVM) algorithm to construct highly-accurate models to predict whether a binary is malicious or not. However, SPGK is computationally expensive due to the size of the input graphs. Therefore, we evaluate different parallelization methods for CPUs and GPUs to speed up this kernel, allowing us to continuously construct up-to-date models in a timely manner. Our hybrid implementation, which leverages both CPU and GPU, yields the best performance, achieving up to a 14.2x improvement over our already optimized OpenMP version. We compared our generated graph-based models to previously state-of-the-art feature vector 2-gram and 3-gram models on a dataset consisting of over 22,000 binaries. We show that our classification accuracy using graphs is over 19% higher than either n-gram model and gives a false positive rate (FPR) of less than 0.1%. We are also able to consider large call graphs and dataset sizes because of the reduced execution time of our parallelized SPGK implementation.

I. Introduction

If one attends any cybersecurity conference or security vendor showcase, it seems as if the industry is in agreement that data breaches cannot be stopped. They say it is not a matter of "if" a company will be breached, but "when" it will be breached. One major reason for the seemingly unstoppable data breaches is that security companies have failed to deliver products that detect and block all malware. Data breaches have become prevalent because bad actors have embraced automation to construct malware. Automation enables bad actors to create so much malware that it is estimated that tens of thousands of new malware variants are being created every hour [1]. In contrast, most security companies that develop products to detect malware still construct them manually [20]. This antiquated method of constructing malware detection systems cannot keep up with the massive amounts of new malware variants created every day.

The work presented in this paper aims to fundamentally change the way in which malware detection engines are constructed. This paper proposes to replace traditional hand-crafted malware detection rules with a self-tuning malware detection engine that adapts its detection rules automatically to match the characteristics of the latest targeted attacks. This will dramatically shorten the cycle from malware discovery to malware rules construction and deployment. Our system involves using graph-based compiler representations of binaries and analyzing these graphs with machine learning algorithms called graph kernels that take as input graphs. These graphs can be extremely large and complex as shown in Figure 1. Thus, while graph kernel algorithms can be effective at learning the subtle differences between goodware and malware, they are computationally expensive because the size of the graphs that we are evaluating. Therefore, we need to parallelize these algorithms in order to make them viable.

The first step of our automatic-tuning malware detection engine consists of using Radare2 [1] to extract assembly code from decompiled binaries. We then use this assembly code to build function call graphs for each application, and we use our parallelized graph kernels to examine the similarities between these call graphs. Specifically, we propose a hybrid parallel implementation of the Shortest Path Graph Kernel (SPGK) that makes use of both the CPU and Graphics Processing Unit (GPU) to efficiently perform these comparisons. The output of the SPGK is a similarity matrix that can be used as input to a Support Vector Machine (SVM) algorithm, which builds the classifier. This classifier is used to identify whether a given application is malicious, and if it is malicious, it identifies the family of malicious software it is a variant of. This classifier can be retrained regularly in order to ensure the classifier utilizes the latest known malware. Our implementation can achieve a classification accuracy of 92% and performance that is up to 14x faster than the optimized OpenMP implementation. We evaluate our framework by

[1]RADARE2: http://www.radare.org/r/

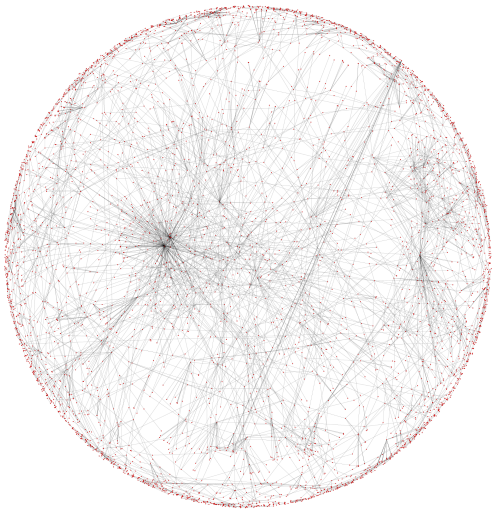


Fig. 1: The graph above corresponds to the call graph of the malware SmsAgent, which is a Java program that targets Android mobile devices. This malware has 20K functions, but we only show the 7K connected functions.

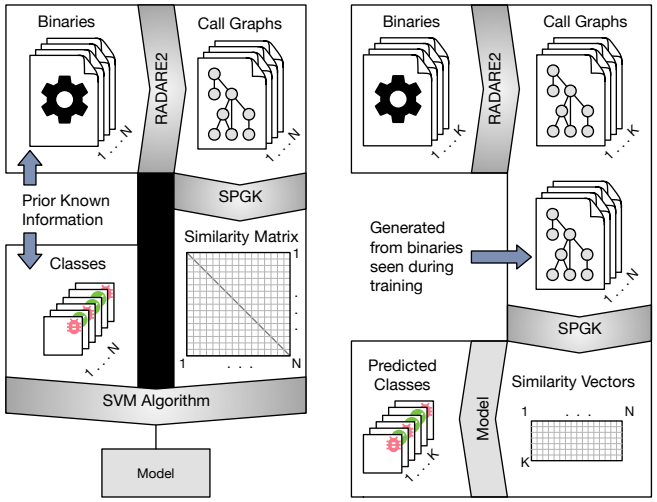
examining both the classification accuracy on different sized datasets and the performance of various different CPU and GPU parallel implementations of SPGK. We show that using a hybrid implementation that takes advantage of parallelism on both the CPU and GPU leads to the best performing implementation. Our hybrid implementation of SPGK allows us to examine large datasets and call graphs, yielding high classification accuracy and very low false positive rates. The main contributions of this paper are the following:

1) We present a highly-accurate and adaptive malware detection engine that leverages a support vector machine to classify binaries based on call graphs constructed from the reverse-engineered assembly code.
2) We show that using graph-based representations and machine learning algorithms that take graphs as input achieves over 19% higher accuracy in detecting malware than traditional methods that focus primarily on flat feature vectors.
3) We accelerate the creation and use of the SVM's classification model using a hybrid (CPU + GPU) parallel implementation of the Shortest Path Graph Kernel.

The rest of this paper is outlined as follows: Section II describes our proposed platform which uses a machine learning algorithm to classify applications, and Section III discusses the call graph representation we use to analyze the decompiled code extracted from binaries. Section IV describes various parallel CPU and GPU implementations of the Shortest Path Graph Kernel (SPGK) used to construct the kernel matrix that the SVM uses to construct a model. Section V describes our experimental framework and presents our results. Section VI discusses related work, and we conclude in Section VII.

II. Malware Analysis of Graphs Involving Calls

In this section, we present the Malware Analysis of Graphs Involving Calls (MAGIC) model. We discuss how a dataset of binaries are decompiled and represented as call graphs, how we use our Shortest Path Graph Kernel (SPGK) to detect similarities in these call graphs, and how the resulting kernel matrix plays a role in generating our machine learning classification model. The model produced during training, as shown in Figure 2a, allows us to classify a program as malicious or benign. An extension of this model allows us to classify the family of malware.



(a) Training Phase (b) Testing Phase

Fig. 2: Figure 2a shows the workflow toward construction of a machine learning model generated during MAGIC's training phase. Figure 2b shows a flow diagram demonstrating the classification of an unseen binary application used during MAGIC's evaluation phase.

A. Feature Extraction and Representation

When monitoring a system for malicious activity in real time, source codes of running applications are generally not available. In most cases, applications are simply downloaded and run as binary executables. Therefore, there is a need to decompile binaries for further analysis. In this paper, we use Radare2, a cross-platform reverse engineering framework that is capable of disassembling binaries that run on many different architectures. As shown in Figure 2, Radare2 produces a call graph of the application's functions, and it extracts the assembly code for each of these functions. This allows us to examine structural qualities of an application, i.e., the caller-callee relationship of the application's functions, as well as the types of instructions used in each function. One major difference between conventional call graphs and the way we present call graphs is the way nodes are labeled. Each function is represented by a feature vector corresponding to a histogram of all the instructions in the function. These feature vectors allow us to represent the total number of instructions in a given function call.

Combining the feature vectors and call graphs gives us a very expressive representation of an application, as well as its content. We then use machine learning algorithms, called graph kernels, to construct similarity scores between pairs of call graphs. Specifically, we use the Shortest Path Graph Kernel to compare our call graphs.

B. Shortest Path Graph Kernel

A key component in our framework is the SPGK algorithm shown in Figure 2. The Shortest Path Graph Kernel (SPGK) algorithm is straight-forward. It takes as input a collection of graphs and determines how similar they are to each other. The output of this algorithms is a kernel matrix, which corresponds to the pairwise similarity values of each pair of graphs in the dataset. In order to run SPGK on a graph, we must first use the Floyd-Warshall algorithm to convert the graph into a fully connected, all-pairs shortest path graph. Since the computation of the similarity of each pair of graphs are not data dependent, we can greatly accelerate this algorithm using parallelism, as discussed in Section IV. Since the output of this graph kernel will be used as the input to an SVM algrithm, parallelism will be integral in enabling us to regularly train on the latest known malware [8]. This will keep the model current.

III. Compiler Representation

Our framework differs from traditional malware detection systems because it leverages structured representations of binaries that are obtained through disassembly. More precisely, our framework utilizes an open-source disassembler, Radare2, to create an high-level representation of the code contained in the binary. Based on this representation, MAGIC constructs a call graph (CG) of the binary which is then labeled based on a histogram of instructions in each function.

A. Disassembly with Radare2

Given a executable file, Radare2 produces a list of routines, where each routine is a list of blocks and each block contains a list of instruction. In addition to the offset, opcode, and operands, Radare2 associates to each instruction one of 53 categories, listed in Table I.

switch	case	call	ucall	jmp	ujmp	cjmp	ucjmp
uccall	ccall	ret	cret				
mov	cmov	swi	length	cmp	acmp	add	sub
abs	mul	div	shr	shl	cpl	sal	sar
or	and	xor	crypto	nor	not		
lea	xchg	ror	rol	mod	cast		
leave	store	load	upush	pop	push	new	io
null	nop	unk	trap	ill			

TABLE I: This table shows the 53 different categories of instructions extracted by Radare2. There are four major categories that the extracted instructions correspond to. In ascending order they are grouped by control flow, arithmetic, memory, and miscellaneous.

B. Construction of the Call Graph

Formally, a call graph (CG) can be represented as $G = \langle V, E \rangle$, where V is a set of nodes and each node $v \in V$ represents one of the functions. $E \subseteq V \times V$ denotes the directed edges, where an edge $e_{i,j} = (v_i, v_j)$ represents a call from the caller function represented by v_i to the callee function represented by v_j. In our CG representation, each vertex is labeled with a feature vector representing a histogram of the instructions in the function. Figure 3 shows the transformation flow from binary representation (assembly) to our graph-based representation with extracted feature vectors for each function.

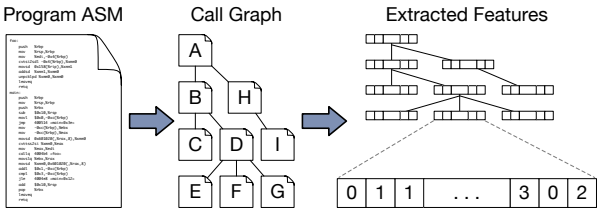


Fig. 3: This figure shows the workflow from binary representation to the graph-based representation with extracted features. An input program is processed through Radare2 to generate grouped assembly instructions. Our analysis parses the Radare2 output to extract histograms of instructions per function, then outputs the feature vectors and the adjacency matrix of the call graph.

The assembly provided by Radare2 is analyzed to build a CG. In the call graph, the nodes represents the binary's functions, while the directed edges represent the caller-callee relationship between functions. For each functions in the assembly, we count the number of occurrences of each category of instruction (see Table I). This form a histogram of 53 elements for each function, this histogram is used to label the corresponding node.

IV. Graph Kernel Parallelization

In order to develop an effective malware detection engine, we contend that using graphs to represent and analyze the structure of applications is more effective than previous state-of-the-art methods. Most prior methods of malware detection only leverage flat feature vectors and do not consider the graph-based structure of applications [2], [4], [7], [12], [17]. Previous work has shown that using graph-based representations of data as input to machine learning algorithms yields more accurate classification results compared to the use of flat feature vectors [14].

A. Background

In this section, we focus on accelerating the Shortest Path Graph Kernel (SPGK), as originally proposed by Borgwardt et al. [6]. Research has shown that it is highly competitive in terms of accuracy and running time, when compared with other kernel algorithms [19]. To the best of our knowledge, no other work has addressed the parallelization of graph kernels when applied to the problem of malware detection. Note that the sequential version of this algorithm runs in $O(n^4)$, which makes it practical only for small graphs. Parallelization will allow us to utilize SPGK to process large graphs, and large datasets of graphs, in a reasonable amount of time enabling continuous training on current data.

B. Shortest Path Graph Kernel

The Shortest Path Graph Kernel (SPGK) was proposed by Borgwardt and Kriegel [6]. This kernel counts the number of shortest paths of the same length having similar start and end vertex labels in two input graphs. One of the motivations for using this kernel is that it avoids the problem of "tottering" found in graph kernels that use random walks [13]. Tottering is the act of visiting the same nodes multiple times thereby artificially creating high similarities between the input graphs. In shortest path kernels, vertices are not repeated in paths, so tottering is avoided.

Given a graph $G = \langle V, E \rangle$, a shortest path graph is a graph $S = \langle V', E' \rangle$, where $V' = V$ and $E' = \{e'_1, \ldots, e'_m\}$ such that $e'_i = (u_i, v_i)$ if the corresponding vertices u_i and v_i are connected by a path in G. Each edge in the shortest path graph is labeled with the shortest distance between the two nodes in the original graph.

Once the shortest path graph is computed for each of our graphs, we can use each shortest path graphs to compute similarity between two graphs using the Short Path Graph Kernel (SPGK) algorithm. SPGK for two shortest path graphs $S_1 = \langle V_1, E_1 \rangle$ and $S_2 = \langle V_2, E_2 \rangle$ is computed as follows:

$$K_{SPGK}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2) \qquad (1)$$

where k_{walk} is a kernel for comparing two edge walks. The edge walk kernel k_{walk} is the product of kernels on the vertices and edges along the walk. It can be calculated based on the start vertex, the end vertex, and the edge connecting both. Let e_1 be the edge connecting nodes u_1 and v_1 of graph S_1, and e_2 be the edge connecting nodes u_2 and v_2 of graph S_2. The edge walk kernel is defined as follows:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \qquad (2)$$

where k_{node} and k_{edge} are kernel functions for comparing vertices and edges, respectively. in the following sections.

Pseudocode for a naive implementation of the Shortest Path Graph Kernel is presented in Algorithm 1. Given two input graphs $g1$ and $g2$, lines 2-7 loop over the shortest path matrices to find all pairs of paths. Line 8 calculates the k_{edge} and lines 10-11 calculate k_{node}. Line 12 calculates k_{walk} and computes the summation.

C. Fast Computation of the Shortest Path Graph Kernel

An implementation of the Shortest Path Graph Kernel (Algorithm 1) has three issues that slow down its performance. First, four for loops and two if statements slow down the algorithm's performance. Second, there is potential redundant computation performed by k_{node}. Third, there is a drawback in the random memory access pattern in Algorithm 1. Sequential memory access is preferred on the CPU and, especially for SIMD architectures like the GPU.

To address the issues of Algorithm 1, we propose a novel way to calculate the shortest path graph kernel. We refer to this as the Fast Computation of Shortest Path Graph Kernel ($FCSP$). In this method, the calculation of the shortest path graph kernel is divided into two main components: 1) calculating all possible instances of k_{node} into a vertex kernel matrix and 2) calculating all required values for k_{walk}. Note that the kernel functions k_{node} and k_{edge} used to calculate the similarity between a pair of nodes and a pair of edges can be different from application to application. In our experiments, we use the Gaussian kernel 4 and the Brownian Bridge kernel 3, which are positive semidefinite [18].

Algorithm 1 Shortest Path Graph Kernel Algorithm

```
1:  K ← 0
2:  for i, j = 0 → n_node[g1] do
3:      w1 ← sp_mat[g1][i][j]
4:      if i ≠ j AND w1 ≠ INF then
5:          for m, n = 0 → n_node[g2] do
6:              w2 ← sp_mat[g2][m][n]
7:              if m ≠ n AND w2 ≠ INF then
8:                  k_edge ← EdgeKernel(w1, w2)
9:                  if k_edge > 0 then
10:                     k^1_node ← NodeKernel(g1, g2, i, m)
11:                     k^2_node ← NodeKernel(g1, g2, j, n)
12:                     K+ = k^1_node * k_edge * k^2_node
13:                 end if
14:             end if
15:         end for
16:     end if
17: end for
18: return K
```

$$k_{brownian}(e_1, e_2) = max(0, c - |e_1 - e_2|) \qquad (3)$$

$$k_{gaussian}(x, y) = exp(-\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{\sigma}) \qquad (4)$$

For the first component, we call $VertexKernel$, we proceed as follows. Assuming that the order of g_1 is m and the order of g_2 is n, we create a matrix $V_{m \times n}$ for storing the k_{node} values, where every entry is the value of $k_{node}(u, u')$ for u being a node of g_1 and u' being a node of g_2. By using this scheme, the redundant computation of k_{node} is eliminated.

The second component, we call $WalkKernel$, is responsible for calculating k_{walk}, and takes advantage of a new representation of the shortest path adjacency matrix. The new representation is composed of three equally-sized arrays. The length of these arrays is the number of edges in the corresponding matrix. The three arrays store the weight of the edge, the index of the starting vertex, and the index of the ending vertex. This representation is inspired by the formats of storing a sparse matrix on GPUs [5], which can solve the low memory utilization problem for sparse matrices access. By applying this transformation, the two if statements in Algorithm 1 can be removed and four for loops are reduced to two.

The pseudo-code of our new method is presented in Algorithm 2. Given input graphs $g1$ and $g2$, function $Vertex\_Kernel$ calculates all possible instances of k_{node}

sequentially and stores them in a matrix V for later access. Function $Walk\_Kernel$ takes advantage of the three 1D arrays converted from shortest path matrix, which creates more sequential memory access and less branch divergence. It calculates all k_{walk} computation and sums them up as the final similarity between two input graphs.

Algorithm 2 Fast Computation of Shortest Path Graph Kernel

```
1:  function VERTEX_KERNEL
2:      for i = 0 → n_node[g1] do
3:          for j = 0 → n_node[g2] do
4:              V[i][j] ← NodeKernel(g1, g2, i, j)
5:          end for
6:      end for
7:  end function
8:  function WALK_KERNEL
9:      K ← 0
10:     for i = 0 → n_node[g1] do
11:         (x1, y1, w1) ← edge_g1[i]
12:         for j = 0 → n_node[g2] do
13:             (x2, y2, w2) ← edge_g2[j]
14:             k_edge ← EdgeKernel(w1, w2)
15:             if k_edge > 0 then
16:                 k^1_node ← V[x1][x2]
17:                 k^2_node ← V[y1][y2]
18:                 K+ = k^1_node * k_edge * k^2_node
19:             end if
20:         end for
21:     end for
22:     return K
23: end function
```

D. FCSP running on Multi-Core CPU

In our experiments we evaluate the calculation of a kernel matrix from a given input dataset of n graphs. Here, we present two different schemes of $FCSP$ parallelization on multicore CPUs. Both schemes are implemented using OpenMP. In the first scheme, $OpenMP\_Graph$, we parallelize the $FCSP$ computation for a single pair of graphs. The second scheme, $OpenMP\_Matrix$ parallelized the kernel matrix calculation.

E. FCSP running on the GPU

$FCSP$ is a suitable application for parallelization. In FCSP, branches are removed, no load balancing issue exists between GPU threads, and the coalesced memory access is satisfied. We are therefore able to achieve significant speedups with this approach.

In our GPU implementation, the $FCSP$ is divided into three GPU kernels. The first one is $Vertex\_Kernel$. It calculates all possible instances of k_{node} and stores them in a matrix for later access. The second kernel is $Walk\_Kernel$, which calculates all the required values for k_{walk} and stores them in a matrix or array. The last component is $Reduction\_Kernel$, which sums up all k_{walk} values into a small array which is copied to CPU memory and summed up as the final similarity.

For the first component, named $Vertex\_Kernel$, we proceed as follows. Assuming that g_1 has m vertices and g_2 contains n, we allocate a buffer $V_{m\times n}$ on the GPU memory for storing the k_{node} value. A GPU thread grid is created, where each thread calculates an entry of V. Since we remove the divergence, all threads in this component are running in parallel.

The second component, named $Walk\_Kernel$, is responsible for calculating k_{walk}. Given two input graphs, suppose the number of paths in g_1 is a and g_2 has b paths. We assume g_1 has more paths than g_2 without a loss of generality. For the graph with n nodes, the paths can vary from 0 to n^2. Therefore, the domain decomposition for GPU threads can be challenging. In our implementation, we assign a GPU thread to one path in g_1. This thread will loop through all the b paths in g_2, calculate the corresponding k_{walk} values, and sum them up. An array of a elements will be returned at the end. The calculation of k_{walk} requires k_{node}, that has already been calculated and cached.

The third GPU kernel is $Reduction\_Kernel$. A reduction is performed on a elements generated by the $Walk\_Kernel$. Upon completion, a small resulting array is copied back to the CPU. Finally, the similarity between the graphs is calculated by adding up all the values in the array.

The biggest advantage of parallelizing $FCSP$ on the GPU is efficiency. There is no execution divergence between threads because of the shortest path matrix conversion in $Vertex\_Kernel$ and $Walk\_Kernel$. The sequential coalesced memory access is satisfied in all three kernels.

In our GPU implementation, the last kernel is the $Reduction\_Kernel$. A small array is copied to the CPU and summed up for calculating the final similarity. This memory copy from GPU to CPU and computation on CPU may not require much time. However, given n input graphs, the GPU method needs to be called n^2 times. As a result, there may be considerable time spent on memory transfers and CPU computation. Our experiments show that the portion of time spent on the reduction memory transfer can vary from 6% to 50% of the total computation, as illustrated in Figure 4. Fortunately, this part can be hidden by overlapping it with GPU computation. When the reduction kernel completes, we initiate a non-blocking memory transfer. We then assign another pair of graphs to the $Vertex\_Kernel$. As the memory transfer is asynchronous it can be overlapped with the next $Vertex\_Kernel$ execution. When $Vertex\_Kernel$ completes we initiate a non-blocking execution of $Walk\_Kernel$ and the CPU accumulates the reduction result array to obtain the similarity result while the GPU is executing. Our experiments show this scheme can hide most of the time spent due to memory transfers.

F. Hybrid Implementation - Combining CPU and GPU

From our experiments, we observed that the implementation with the best performance varied depending on different datasets. When the graphs were small, the CPU implementations beat all the GPU implementations. However, when the graphs are large, the GPU performed best. The experiments show that the computation/communication overlapped implementations always performed better than the ones without overlapping. Combining a CPU/GPU implementation seemed

	Nodes			Edges		Shortest Path	
	Min	Max	Avg	Max	Avg	Max	Avg
6K	10	19	13	28	11	60	14
12K	10	44	20	139	16	205	23
21K	10	399	56	2,542	45	25,311	110
22K	10	4,841	81	2,615	86	62,961	371

TABLE II: This table shows the statistics for our four datasets. The first column gives the approximate data set size. The second and third columns show statistics for the nodes and edges of the original graphs. The fourth column shows the statistics for paths in the shortest path graphs.

to be a good idea. We hypothesize that many real world datasets have graphs of a variety of different sizes whose processing can be improved by partitioning the datasets to take advantage of the heterogeneous environment. So in our $Hybrid$ implementation, we first set a threshold $T1$ for average graph size in the input dataset. If the average graph size is smaller than $T1$, then we use $OpenMP\_Matrix$ to calculate the full kernel matrix. Otherwise, we set another threshold $T2$ for graph size to decide graphs that should be run on the CPU versus the GPU. When the number of shortest paths in both input graphs are smaller than $T2$, we use $OpenMP\_Graph$ to calculate the similarity. Otherwise, the $GPU\_overlap$ is used. Additionally, in our $Hybrid$ implementation we compress the input graphs using the Compressed Row Storage (CRS) format because we observe that the labels in our graphs are usually sparse. By adapting the CRS format, we are able to significantly reduce the average dimension of vertex labels by eliminating all zero elements. Consequently, the computation time is greatly decreased.

V. Evaluation and Results

This section discusses the datasets and the computational performance results and machine learning accuracy of our different implementations.

A. Binary Sources

We have 37,176 malicious applications classified using Reversing Labs' [16] A1000 Cloud Analysis and 14,706 instances of benign applications (goodware). The benign applications were collected from a Windows 10 system where the 300 most popular applications available from Chocolatey [15], a command-line based application installer, were installed. These binaries are processed by MAGIC's pipeline (Section II). MAGIC utilizes Radare2 to extract the call graph of each application.

Our goal is to perform the construction of machine learning models in under a few hours. We determined thresholds to build datasets of call graphs that meet this criteria. SPGK's complexity is the product of the number of shortest paths in each graph. However, the all-pairs shortest path problem is solved in $O(V^3)$ in the general case, where V is the number of nodes. Because we want the selection of graph to be fast, we only consider the number of nodes and the number of edges for our selection criteria.

In order to evaluate the scalability of our proposed model, we measured its performance on datasets of varying sizes.

	GPU 1D		OpenMP		Hybrid
		Overlap	Matrix	Graph	
6K	1.58 h	40.52 m	1.01 m	3.19 m	4.65 s
	0.01x	0.03x	1.0x	0.35x	14.2x
12K	9.36 h	5.61 h	10.96 m	20.48 m	52.57 s
	0.02x	0.03x	1.0x	0.53x	12.5x
21K	memory exhausted		4.55 h	5.50 h	50.37 m
			1.0x	0.83x	5.42x
22K			6.51 h	12.24 h	2.47 h
			1.0x	0.53x	2.63x

TABLE III: Comparing similarity matrix computation time for each dataset. Runtimes (h: hours, m: minutes, s: seconds) are presented for each implementation. Below each runtime we give speedup compared to the best OpenMP (OpenMP Matrix) implementation is shown. VRAM limitation was exceeded for our larger datasets when only running on the GPU.

Table II presents statistics about the four datasets we constructed. They are labeled based on the approximate number of call graphs they contain. The dataset was constructed with the following constraints:

- 6K dataset – nodes ≥ 10 and nodes < 19
- 12K dataset – nodes ≥ 10 and nodes < 45
- 21K dataset – nodes ≥ 10 and nodes < 400
- 22K dataset – nodes ≥ 10 and edges < 3000

For the last dataset we used a limit on the number of edges instead of the number of nodes because we were reaching the point at which the average number of shortest-paths (which our selection method is not aware of) was becoming too large for our implementation to handle under the time constraint.

B. Computational Performance Results

We evaluated our framework on a heterogeneous architecture machine with an Intel i7-5860K CPU, 32GB DDR4 @ 2133MHz, and dual NVIDIA GTX 970s. Table III presents the runtime of different implementations of SPGK for the four datasets. Figure 4 illustrates GPU utilization while processing the 6K and 12K datasets. It was not possible to process the largest datasets (21K and 22K) using the GPU version of SPGK because of memory exhaustion. Despite these limitations, our hybrid implementation outperforms the OpenMP implementation by considering the GPU for certain computations. This hybrid implementation offloads computation to the GPU if there are more than 125 shortest path edges in the graphs that are being compared. The GPU implementation never performs as well as any OpenMP implementation because of resource underutilization. The GPU is underutilized for any graph that has a shortest path edge count smaller than the average number of shortest paths for all graphs. There is not enough computation to be done for each thread to take advantage of the massive gains achievable with the increased throughput of the GPU. The overlap version of the GPU implementation helps reduce the execution time by up to a factor of 14.2x.

The $OpenMP\_Graph$ implementation does not perform as well as the $OpenMP\_Matrix$ implementatio. By making use of both the CPU and the GPU, the hybrid implementation achieves a speed up of 2.63x on our largest dataset when

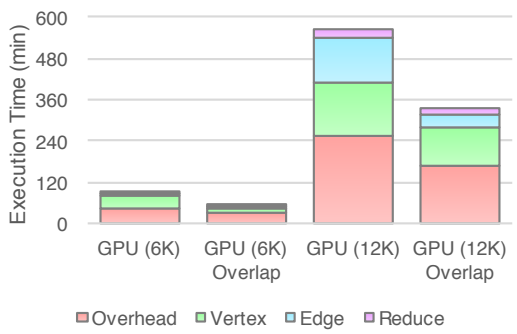


Fig. 4: This figure highlights the execution time comparison for the GPU executions of the 6K and 12K datasets. The overlap kernels outperform their corresponding original implementations by overlapping data transfer with computation. The implementation's ability to scale well based on the dataset size is not supported with the execution results – the 21K and 22K dataset is unable to be run only on the GPU due to memory exhaustion.

Features	Metric	Number of Classes 2	33
SPGK	Accuracy	91.02%	90.80%
	FPR	00.15%	00.24%
	FNR	20.08%	20.53%
2-grams	Accuracy	70.31%	69.65%
	FPR	13.59%	12.96%
	FNR	49.94%	54.38%
3-grams	Accuracy	71.99%	70.71%
	FPR	10.75%	11.43%
	FNR	49.71%	54.16%

TABLE IV: This table gives accuracy, false positive, and false negative rates for the different characterization types. We consider either two classes (goodware and malware) or multiple classes (goodware and types of malware).

compared to the best OpenMP implementation. This efficient runtime is possible due to the simple heuristic based on the shortest path size to determine which implementation to use, and it allows us to train our classifier on larger sets of data given a fixed time frame, ideally resulting in a more accurate model. It is worth noting that increased bandwidth is necessary for comparing larger graphs.

C. Machine Learning Accuracy

After model construction, we feed the resulting graph similarity matrix into a SVM learning algorithm to generate a model. We evaluate how our model performs when characterizing executable binaries for the purpose of detecting and identifying malware. We performed the machine learning experiments on the 22K dataset.

1) Baseline: We compare our accuracy results against N-grams of the bytes in the binaries. N-grams are histograms of all the sequences of N bytes in the binary. We consider 2-grams and 3-grams of the bytes of each binary. In both cases, we build a feature vector of the 10,000 most frequent N-grams. These feature vectors are then used to build an SVM model ($C = 100$, Radial Basis Function kernel with $gamma = 100$).

2) Accuracy: In Table IV, we present the summary of our machine learning experiments. We evaluated all three set of features for two classification problems. The first problem uses the model to classify binaries as either goodware or malware without distinction between different strains of malware. The second problem requires the model to classify binaries as goodware or one of the 32 different strains of malware in the dataset.

For each type of model we used 5-folds cross-validation (training on 80% of the dataset and testing on 20% of the dataset, five times to have each fold used as test). We use three metrics to evaluate the resulting model: classification accuracy, number of false positives, and number of false negatives [23].

- Accuracy: Percentage where binaries are correctly identified as goodware or malware.
- False Positive Rate (FPR): rate where goodware is predicted as malware
- False Negative Rate (FNR): rate where malware is predicted as goodware

These three metrics tell us different stories, and models can be tuned to favor the desired metric. In the case of malware detection, accuracy is the primary metric, but minimizing the number of false positives is also important because we if we flag too many good applications as malicious this can be disruptive to the user. Additionally, too many false positives can cause the user to dismiss the detection of real malware.

Table IV shows that comparing binaries using call graphs and the shortest path graph kernel outperforms 2-gram and 3-gram of the bytes of the binaries. First, we look at the two classes problem. The accuracy of the SPGK model is 19.03% greater than 3-gram (which outperforms 2-gram by 1.68%). Second, the SPGK model has a FPR more than 2 order of magnitude lower than 2-gram and 3-gram on the same problem. Third, the accuracy of the 2-gram and 3-gram models decrease by 0.66% and 1.28% respectively when applied to the multi-classes problem compare to the two-classes problem. By comparison, the SPGK model only loses 0.22% accuracy when working on the multi-classes problem instead of the two-class model.

VI. Related Work

In this section, we examine previous works on malware detection strategies that make use of various compiler representations and techniques (particularly graphs), and graph kernel parallelization.

A. Graph-Based Malware Detection

Solving the problem of malware detection by examining structural and behavior qualities of applications can be seen as a compiler techniques issue. In most cases, source code is not available, so some type of decompiler is required. Additionally, compiler representations and techniques can be used to analyze an application's decompiled code.

Yang et al., developed DroidMiner [22] which uses static analysis to automatically mine malicious program logic from known Android malware. Behavior graphs are constructed from malware in DroidMiner, and these graphs are flattened into feature vectors that are then fed into several machine

learning classifiers including naive Bayes, SVM, decision trees, and random forests for malware detection. The best algorithm of DroidMiner can achieve a 95.3% detection rate on a dataset of 2466 malware. It can also reach 92% for classifying malware into its proper family.

Anderson et al. [2] and Ak-Bakri et al. [3] proposed algorithms for malware detection that make use of graph-based representations of instruction traces of binaries. Each graph represents a Markov Chain, where the vertices represent instructions. They use a combination of different graph kernels to construct a similarity matrix between these graphs. Like us, they feed this resulting similarity matrix to an SVM to perform classification. These papers does not address the possibility of optimizing or parallelizing these algorithms, which exceeds $O(n^2)$.

Gascon et al., proposed a method for malware detection based on efficient embedding of Function Call Graphs (FCG), which are high level characteristics of the applications [10]. They extracted function call graphs using the Androguard framework [9]. The nodes in the graph were labeled according to the type of instructions contained in their respective functions. A neighborhood hash graph kernel was applied to evaluate the count of identical substructures in two graphs. Finally, an SVM algorithm was used for classification. In an evaluation of 12158 malware samples, the proposed method detected 89% of the malware. We contend that our paper presents a similar framework that achieves higher accuracy and yields better performance due to our parallel implementation of the graph kernel used to construct the similarity matrix that is fed into the SVM.

B. Parallelization of Graph Kernels

There is a limited amount of research available that focuses on parallel implementations of graph kernels. Hong et al. present a method for implementing a parallel version of breadth-first search (BFS) and present results on both multi-core CPU and GPU [11]. They also present a hybrid method which dynamically chooses which of their implementations will yield the best performance during each BFS iteration. Although the kernel itself is different, this work shows a viable hybrid parallel implementation of a graph traversal algorithm which scales well when operating on large graphs. The hybrid implementation of SPGK [21] is similar in nature, but to the best of our knowledge, this paper is the first to leverage a hybrid implementation of a graph kernel for malware detection.

VII. Conclusion

We use the Shortest Path Graph Kernel (SPGK) to identify similarities between call graphs extracted from binaries. However, SPGK is computationally expensive due to the size of the input graphs. Therefore, in this paper we evaluate different parallelization methods for CPUs and GPUs to speed up this kernel. Our hybrid parallel implementation yields the best performance, achieving up to a 14x improvement over the baseline OpenMP version. We evaluate our models on increasing data set sizes to evaluate the scalability of our different parallel implementations.

For future work, we plan to evaluate further optimizations to improve the efficiency of our parallel implementations. We plan to investigate partitioning the largest graphs into tiles so that the subset of the graph's nodes can be sent in a staged fashion to avoid using all device memory. There are additional optimizations involving the ordering of nodes being compared that can improve the efficiency of our implementation. Finally, we plan to investigate different graph kernel algorithms besides the SPGK which may lead to more efficient parallel implementations with similar or improved accuracy.

References

[1] "A brief history of malware," in *Webroot Threat Blog*, 2015.

[2] P. A. M. Al-Bakri and H. L. Hussein, "Static analysis based behavioral api for malware detection using markov chain," *The International Institute for Science, Technology and Education (IISTE)*, vol. 5, 2014.

[3] B. Anderson *et al.*, "Graph-based malware detection using dynamic analysis," *J. Comput. Virol.*, vol. 7, no. 4, pp. 247–258, Nov. 2011. Available: http://dx.doi.org/10.1007/s11416-011-0152-x

[4] S. S. Anju *et al.*, "Malware detection using assembly code and control flow graph optimization," in *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, ser. A2CWiC '10. New York, NY, USA: ACM, 2010, pp. 65:1–65:4. Available: http://doi.acm.org/10.1145/1858378.1858443

[5] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[6] K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," in *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2005. Available: 74-81

[7] M. Christodorescu *et al.*, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*, May 2005, pp. 32–46.

[8] N. Cristianini and J. Shawe-Taylor, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[9] A. Desnos, "Androguard - reverse engineering, malware and goodware analysis of android applications," 2011. Available: http://code.google.com/p/androguard/

[10] H. Gascon *et al.*, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec)*, 2013.

[11] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 78–88.

[12] J. D. Igor Santos, Yoseba K. Penya and P. G. Bringas, "N-grams-based file signatures for malware detection," in *Proceedings of the 11th International Conference on Enterprise (ICEIS)*, 2009.

[13] P. Mahé *et al.*, "Extensions of marginalized graph kernels," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2004, pp. 552–559.

[14] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 196–206. Available: http://doi.acm.org/10.1145/2259016.2259042

[15] Real Dimensions Software. (2016) Chocolatey machine package manager. Available: https://chocolatey.org

[16] Reversing Labs. (2016) Reversing Labs. Available: http://reversinglabs.com

[17] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct 2015, pp. 11–20.

[18] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[19] N. Shervashidze and K. Borgwardt, "Fast subtree kernels on graphs," in *Proceedings of the Neural Information Processing Systems Conference (NIPS)*, 2009, pp. 1660–1668.

[20] M. Sikorski and A. Honig, "Practical malware analysis: The hands-on guide to dissecting malicious software (1st ed.)," in *No Starch Press*, 2012.

[21] L. Xu *et al.*, "Parallelization of shortest path graph kernels on multi-core cpus and gpus," in *Programmability Issues for Heterogeneous Multicores (MultiProg '14)*, 2014.

[22] C. Yang, "Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Computer Security - ESORICS 2014, Lecture Notes in Computer Science. 2014*, 2014.

[23] A. Zheng, *Evaluating Machine Learning Models: A Beginner's Guide to Key Concepts and Pitfalls*, 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 2015. Available: http://www.oreilly.com/data/free/files/evaluating-machine-learning-models.pdf