Using GPUs to Crack Android Pattern-based Passwords

Jaewoo Pi   Pradipta De   Klaus Mueller
Dept. of Computer Science
SUNY Korea
Email: {jwpi, pradipta.de, mueller}@sunykorea.ac.kr

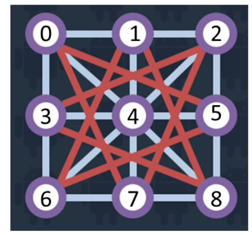
***Abstract*—We investigate the strength of patterns as secret signatures in Android's pattern based authentication mechanism. Parallelism of GPU is exploited to exhaustively search for the secret pattern. Typically, searching for a pattern, composed of a number of nodes and edges, requires an exhaustive search for the pattern. In this work, we show that the use of GPU can speed up the graph search, hence the pattern password, through parallelization. Preliminary results on cracking the Android pattern based passwords shows that the technique can be used as the basis to implement a tool that can check the strength of a pattern based password and thereby recommend strong patterns to the user.**

I. Introduction

Android operating system for mobile phones has introduced pattern based authentication mechanism. The pattern consists of an arbitrary number of strokes or lines between nine dots on the screen. Specifically the Android pattern locking scheme provides an onscreen 3x3 grid of contact points. A pattern is an ordered list of points, without violating three rules: (i) the pattern must contain at least 4 points, thereby any pattern must have at least 3 edges, (ii) a contact point can be used only once, and (iii) intermediate points are selected automatically as a contact point, unless it is already part of an edge in the pattern. Due to these restrictions, total number of allowed patterns on an Android device is 389,112 [1].

Despite its advantages, patterns are easier to hack. Known techniques include shoulder surfing, where the password can be easily remembered by the hacker [2]. Smudge attack is another known technique to steal the pattern password, where the finger traces left on the screen can be used to detect the password [1]. A recent work shows that picture gesture authentication(PGA) scheme used on Microsoft 8 surface devices are hackable by guessing the user choices [3]. Are patterns easy to guess or discover? For alphanumeric passwords, there are tools which recommend if the combination of characters is vulnerable to cracking, either by brute force attack or dictionary based attacks [4]. We investigate whether similar attack to crack the pattern based password on Android devices is feasible or not.

Password patterns are graphs with a vertex count between 4 to 9, and maximum 8 edges. A brute force search is equivalent to looking for all possible patterns. Whereas searching for commonly used patterns and converging on a match faster is analogous to dictionary based attack. We focus on the brute force search problem. However, instead of using CPU to explore for the pattern, we recognize that the search can be executed in parallel using GPUs, where each thread of execution traverses a different search path. In the following sections, we present our technique for parallelizing the graph search function on General Purpose GPUs to crack the Android pattern password, as well as, present some preliminary results showing the vulnerability of the pattern based password.



1	3	4	5	7	-1	-1	-1
0	2	3	4	5	6	8	-1
1	3	4	5	7	-1	-1	-1
0	1	4	6	7	2	8	-1
0	1	2	3	5	6	7	8
1	2	4	7	8	0	6	-1
3	4	7	1	5	-1	-1	-1
3	4	5	6	8	0	2	-1
4	5	7	1	3	-1	-1	-1

Fig. 1. The Android pattern lock grid and the corresponding adjacency matrix denoting the connectivity in patterns

II. Solution Overview

In graph theory, finding a path that visits each vertex exactly once is called Hamiltonian Path Problem. The problem is NP-complete, and the execution slows down exponentially as the number of vertices and edges increases. Traditional method uses divide and conquer approach, which recursively visits all the nodes to find a path. GPUs can run tens of thousands of threads concurrently, however, many GPUs do not support recursive call within each thread in the GPU due to deficiency of kernel stack. Latest GPU release, like Kepler K20, supports recursion facilitated by dynamic branching.

The GPU implementation is optimized progressively to crack the pattern faster. The algorithm recursively traverses neighbor nodes in column order of the adjacency matrix, as shown in Fig. 1. In this matrix each row stands for a node, and each column is connected neighbor, with a maximum possible of up to 8 neighbors. For example, starting from node 0 (left-top) it visits node 1, which is first index of node 0. Then when recursive traversal from $0 \rightarrow 1$ is over, it begins $0 \rightarrow 3$, $0 \rightarrow 4$, and so on. The algorithm prevents revisiting any node in a path by using a simple bit mask data structure.

A. Naive GPU algorithm

The naive approach spawns multiple threads and ensures that each thread explores a different path in the graph. A unique thread id indexes the path traversed. Use octal representation for thread id allows to use each digit as a node in the pattern. For example, a thread whose $tid = 100$ is converted to 144 (in octal). 144 denotes that traversal path is 1st neighbor of starting node, followed by the 4th neighbor of the next node, and the 4th neighbor of the next. In the worst case, a node can have eight neighbors and a path connects up to nine nodes. Thus, without removing any loop or duplicate path it requires $8^8 = 16,777,216$ different paths for each starting node. Threads per block is set to 1024 to reduce number of blocks used, still requiring 16K blocks.

Algorithm 1 CPU PROCEDURE

```
Calculate the number of threads
Allocate CPU and GPU Memory with the number of threads
for startNode = 0 → 8 do
    /* The following function executes on the GPU */
    FindPathKernel(*paths, *adjacencyMatrix, startNode)
end for
```

Algorithm 2 FINDPATHKERNEL()

```
Require: *paths, *adjacencyMatrix, startNode
/* This procedure runs on GPU */
threadID = blockIdx.x * blockDim.x + threadIdx.x
pathOct = threadID
while pathOct > 0 do
    edgeIndex = pathOCT&0b0111
    nextNode = adjacencyMatrix[nextNode][edgeIndex]
    if nextNode refers out of index then
        break;
    end if
    if duplicateCheck[nextNode] == true then
        break;
    end if
    paths = AddNodeToPath(threadID)
    duplicateCheck[nextNode] = true
    pathOct = pathOct >> 3
end while
```

B. GPU Optimization: Thread Reduction

The large number of threads as well as global memory access becomes a key bottleneck. Observe that most threads, except those traversing 9 points, end early. Also, every node has at least one neighbor. For optimization, now the threads only travel one of 7 different neighbors (indexed 1-7) from a node, and threads completing early traverse the neighbor with index 0 after recording their traversal result. Number of total threads are reduced to $7^8 = 5,764,801$ (34.5% of the original) allowing better thread utilization. Also, only the valid paths (ones with no loop in a path) are written to memory. Although number of writes to the memory is reduced, coalescing problem remains unsolved.

C. GPU Optimization: Version 3

Note that each thread has to refer to adjacency (neighbor) matrix at least once and up to eight times. As the matrix does not change while finding the path, it can be written once into constant memory, which is faster than global memory, and supports caching for faster access.

D. GPU Optimization: Version 4

For the indexing from thread ID to path, dec2oct() function is used which converts decimal number into octal number. The function uses division (/) and modular (%) operator leading to slowdown. Replacing modular operation with logical operator AND, and division by eight with 3 right shift operations gives good speedup.

The algorithm for the GPU-based implementation of the traversal is described in Algorithm 1 and Algorithm 2. The procedure starts on the CPU and spawns threads to run on the GPU, using the FindPathKernel function.

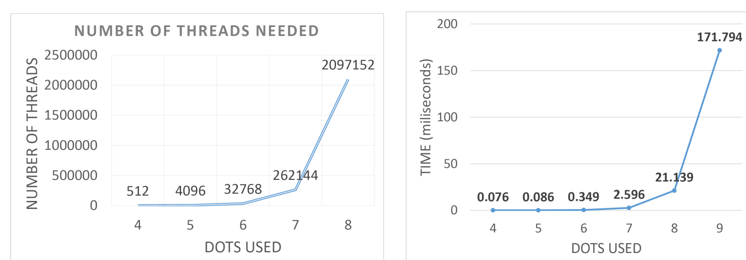
Version	Time(ms)	Speed-up(vs. CPU)	Speed-up (GPU-Naive)
CPU	134	-	-
GPU: Naive	646	x0.21	-
GPU: Thread reduction	100	x1.34	x6.46
GPU: v3	75	x1.79	x8.61
GPU: v4	39	x3.44	x16.56

TABLE I. SPEEDUP IN IDENTIFYING PATTERNS USING GPU

III. PRELIMINARY RESULTS

The preliminary results report the time to crack a pattern. The system used for the experiment is Intel Xeon E5-2630 CPU with 16 GB RAM, and Nvidia Quadro 4000 GPU with 2GB of video memory which has 256 CUDA cores and CUDA 2.1 compute capability. The execution time does not include memory allocation and copy from CPU memory to GPU or vice versa. We show the speedup that GPU processing can achieve over using CPU in Table I. Due to overhead of large number of threads GPU:naive performs worse than execution on CPU, but with further optimizations there is speedup up to 3 times.

Fig. 2(a) shows the number of threads used as the number of contact points increase in a pattern, while the time taken is shown in Fig. 2(b). Note that a pattern with 9 dots, which is the largest pattern allowed on Android, can be cracked in less than 200 ms.



(a) Increasing thread count

(b) Increasing time to crack

Fig. 2. Performance evaluation of pattern based password cracking: With increasing number of contact points in a pattern the threads and the time to crack increases.

ACKNOWLEDGEMENT

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "IT Consilience Creative Program" (NIPA-2013-H0203-13-1001) supervised by the NIPA (National IT Industry Promotion Agency).

REFERENCES

[1] A. J. Aviv, K. Gibson, E. Mossop, M. Blaze, and J. M. Smith, "Smudge attacks on smartphone touch screens," in *Proceedings of the 4th USENIX conference on Offensive technologies*, ser. WOOT'10, 2010.

[2] N. H. Zakaria, D. Griffiths, S. Brostoff, and J. Yan, "Shoulder surfing defence for recall-based graphical passwords," in *Proceedings of the Seventh Symposium on Usable Privacy and Security*, ser. SOUPS '11, 2011.

[3] Z. Zhao, G.-J. Ahn, J.-J. Seo, and H. Hu, "On the security of picture gesture authentication," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[4] D. Apostal, K. Foerster, A. Chatterjee, and T. Desell, "Password recovery using mpi and cuda," in *2012 19th International Conference on High Performance Computing*, 2012.