THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY

DISTRIBUTED BRUTE-FORCE ATTACK IMPLEMENTATION WITH GPGPU ON SQLCIPHER

Master's Thesis

MAHMUT GÜNDEŞ

ISTANBUL, 2019

THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING (ENGLISH, THESIS)

DISTRIBUTED BRUTE-FORCE ATTACK IMPLEMENTATION WITH GPGPU ON SQLCIPHER

Master's Thesis

MAHMUT GÜNDEŞ

Supervisor: ASSIST. PROF. DR. AHMET NACİ ÜNAL

ISTANBUL, 2019

THE REPUBLIC OF TURKEY
BAHCESEHIR UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
COMPUTER ENGINEERING (ENGLISH, THESIS)

Name of the thesis: Distribute Brute-Force Attack Implementation With GpGpu On SqlCipher
Name/Last Name of the Student: Mahmut GÜNDEŞ
Date of the Defense of Thesis: 07.08.2019

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Assist. Prof. Dr. Yücel Batu SALMAN
Graduate School Director

I certify that this thesis meets all the requirements as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Tarkan AYDIN
Program Coordinator

This is to certify that we have read this thesis and we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Science.

Examining Comittee Members	Signature
Thesis Supervisor Assist. Prof. Dr. Ahmet Naci ÜNAL	---------------------------------
Member Assist. Prof. Dr. Serkan AYVAZ	---------------------------------
Member Assist. Prof. Dr. Tayfun ACARER	---------------------------------

ABSTRACT

DISTRIBUTED BRUTE-FORCE ATTACK
IMPLEMENTATION WITH GPGPU ON SQLCIPHER

Mahmut GÜNDEŞ

Computer Engineering (English, Thesis)

Thesis Supervisor: Assist. Prof. Dr. Ahmet Naci ÜNAL

Ağustos 2019, 40 pages

In this thesis, the main topic is discussing SQLCipher brute-force attack in an acceptable resource consuming in acceptable duration. SQLCipher uses AES256 with extra 64000 key iteration to make it more secure.

For this purpose, at first a dictionary file including one hundred billion passwords was generated to be used for brute-force attack. Second, a software was developed to use this dictionary for brute-force in different methods. Software uses different methods to test and get results in order to compare and evaluate. Those methods are:

a. Using single CPU
b. Using multi CPUs on the single machine
c. Using single GPGPU
d. Using multiple GPGPUs on the single machine
e. Using all CPUs on all machines on the network as distributed
f. Using all GPGPUs on all machines on the network as distributed

Third, a graphical user interface developed to monitor and manage software.

Keywords: SqlCipher, Brute-Force, GPGPU, Distributed, AES, PBKDF2

ÖZET

DAĞITIK SİSTEMLERDE GPGPU KULLANARAK BRUTE-FORCE YOLUYLA SQLCIPHER ŞİFRE KIRMA UYGULAMASI

Mahmut GÜNDEŞ

Bilgisayar Mühendisliği (İngilizce, Tezli)

Tez Danışmanı: Dr. Öğr. Üyesi Ahmet Naci ÜNAL

Ağustos 2019, 40 sayfa

Bu tezde AES 256 bit ile şifrelenmiş ve anahtarı 64000 iterasyonla güvenliği arttırılmış bir SQLCipher dosyasının brute-force yoluyla kırılmasında hangi yöntemlerin kullanılabileceği, bu yöntemlerin birbiriyle kıyaslaması ve sonuçta SQLcipher ile şifrelenmiş bir verinin kabul edilebilir bir sürede kırılıp kırılmayacağı görülmek istenmiştir.

Bu amaç doğrultusunda öncelikle karmaşık 32 karakter içeren bir şifre ile şifrelenmiş SQLCipher dosyası ve denenmek üzere yüz milyar şifre içeren bir sözlük oluşturulmuştur. Bu sözlük kullanılarak sırasıyla aşağıdaki metodlar denenecek ve brute-force sonuçları elde edilip karşılaştırılacaktır.

a. Tek işlemci
b. Çoklu işlemci
c. Tek GPGPU
d. Çoklu GPGPU
e. Dağıtık makinalardaki çoklu işlemciler
f. Dağıtık makinalardaki çoklu GPGPU’lar

Elde edilen sonuçlar harcanan kaynak ve gereken zaman açısından gözlemlenerek pratik kullanımının mümkün olup olmadığı tartışılmıştır.

Keywords: SqlCipher, Brute-Force, GPGPU, Distributed, AES, PBKDF2

CONTENTS

TABLES vii
FIGURES viii
ABBREVIATIONS ix
1. INTRODUCTION 1
1.1. AIMS 1
1.2. RESEARCH METHODOLOGY 3
1.3. RESEARCH CONTRIBUTION 3
1.4. THESIS STRUCTURE 4
1.5. SUMMARY 4
2. LITERATURE REVIEW 5
2.1. GPGPU 5
2.2. OPENCL 7
2.3. AES 8
2.4. PBKDF2 10
2.5. SQLCIPHER 11
2.6. SUMMARY 12
3. DATA AND METHOD 13
3.1. DICTINARY CREATION 13
3.2. DICTINARY LOAD METHODS 16
3.3. SQLITE FILE VALIDATION 17
3.4. CPU BASED METHOD 19
3.4.1. Single CPU with SqlCipher Library 20
3.4.2. Multi CPU with SqlCipher Library 23
3.4.3. Single CPU with SSL Library and Buffered Data 25
3.4.4. Multi CPU with SSL Library and Buffered Data 26
3.5. GPU BASED METHODS 27
3.5.1. Single CPU and Single GPU with Buffered Data 27
3.5.2. Multi CPU and Multi GPU with Buffered Data 32
3.6. DISTRIBUTED METHODS 33
3.6.1. Distributed GPUs 34

4. DISCUSSION AND CONCLUSION 38
REFERENCES 41

TABLES

Table 3.1: Possible Character Set For Passwords 14
Table 3.2: Dictionary Processor Header 17
Table 3.3: SQLite File Checker Data 18
Table 3.4: SQLite File Checker Method 19
Table 3.5: SQLite Password Test Method Decleration 21
Table 3.6: SQLite Query Execution Method 21
Table 3.7: Password Processor Worker Method 22
Table 3.8: Multi CPU Usage Threads Join 24
Table 3.9: Iteration and Decryption with SSL Library Method 26
Table 3.10: Input/Output Buffer Preperation for GPU Kernel 29
Table 3.11: Copy Host Memory Buffer to GPU Memory 29
Table 3.12: Kernel Parameters 30
Table 3.13: Work Group/Item Counts 30
Table 3.14: Test Derived Keys 31
Table 3.15: Derived Key Test Method 31
Table 3.16: Distributed Password Processor Main Loop 36
Table 4.1: Methods’ Performance Results 39

FIGURES

Figure 2.1: CPU vs GPU Task Handling 6
Figure 2.2: GPU Memory Model 7
Figure 2.3: CPU vs GPU AES Performance Up To Data Size 9
Figure 2.4: PBKDF2 Performance CPU vs GPU vs FPGA 11
Figure 3.1: Single CPU Usage Test Threads Showing on Htop 23
Figure 3.2: Multi CPU Usage Test Threads Showing on Htop 25
Figure 3.3: Single GPU Controlled With Single CPU 28
Figure 3.4: Performance Test with Single Gpu Using Two Dictionaries 32
Figure 3.5: Multi GPU Controlled with Multi CPU 33
Figure 3.6: Distributed Single/Multiple GPUs on Different Machines 34
Figure 3.7: Server UI, Testing with Six Machines Each Has 8 GPUs 35
Figure 3.8: Server UI, Password Found 37

ABBREVIATIONS

AES	:	Advanced Encryption Standard
CPU	:	Central Processing Unit
DB	:	Data Base
FPGA	:	Field Programmable Gate Array
GPGPU	:	General Purpose computing on Graphics Processing Units
GPU	:	Graphic Processing Unit
HPC	:	High Performance Computing
NIST	:	National Institute of Standard and Technology
OpenCL	:	Open Computing Language
PBKDF2	:	Password Based Key Derivation Function 2
RAM	:	Random Access Memory
SSL	:	Secure Socket Layer
UI	:	User Interface
UUID	:	Unified User IDentifier

1. INTRODUCTION

Encryption is one of the main topics of data security. There are too many algorithms developed for data security and still, cryptographers work on it to move security to a higher level all around the world. On the other hand, there are also works to find the weakness of algorithms and attack types to access encrypted data.

AES is one of the most used symmetric key algorithms in the world and it is still very popular since standardized by NIST in 2001. We use AES in our mobile applications, desktops, servers, and even embedded devices to encrypt files or data over the network. There are too many works on finding the vulnerability of AES and attack tries on AES as well as other encryption algorithms, however, there is not an acceptable way other than a brute-force attack. Fortunately, brute-force attacks on AES are not feasible since it might take days, weeks, months or years up to possible dictionary file size.

SQLCipher is an open-source library and uses AES with 64000 key iterations. SQLCipher is widely used, especially in mobile applications. There are too many famous mobile applications depend on SQLCipher like Snapchat and WeChat.

In this thesis, we try to make a brute-force attack to find the password of a file encrypted by using SQLCipher library (which uses AES internally) with different methods. Used methods are listed below up to feasibility and acceptance.

a. using a single CPU
b. using multi CPUs on the single machine
c. using single GPGPU
d. using multiple GPGPUs on the single machine
e. using all CPUs on all machines on the network as distributed
f. using all GPGPUs on all machines on the network as distribute

1.1. AIMS

This thesis has two main motivations. The first one is to build a software to do brute-force attacks on encrypted files with widely used SQLCipher library. Software will use different methods by using CPUs and GPGPUs in order the find faster and low resource consumer solution. Software consists of a executable running as a daemon and a graphical user interface for monitor and manage executables via network.

The second one is to discuss the security of our data and files since SQLCipher is widely used in mobile applications. SQLCipher is known very secure and used in most famous applications in mobile phones.

This thesis studies brute-force attacks on SQLCipher in six different ways to direct three main objectives:

a. Generating a dictionary file including one hundred billion passwords.
b. Developing a software to make brute-force attack on SQLCipher encrypted file with single CPU, multi CPU, single GPGPU, multi GPGPU, all CPUs in network as distributed and all GPGPUs in network as distributed.
c. Collection results from brute-force attack tries and compare results to discuss SQLCipher security level.

Given the above-mentioned objectives, our research questions are identified as follows:

i. Is it possible to build a dictionary including all possible passwords to be used in brute-force attack?
ii. Is it possible to develop software to test different methods effectively for brute-force attack? In other words.
    a. Is software fast enough while using CPU?
    b. Is software fast enough while using GPGPU?
    c. Is software able to work as distributed to use all CPUs and GPGPUs on network?

iii. Is it possible to collect all data from software brute-force attacks and deduct a result on SQLCipher security level.

1.2. RESEARCH METHODOLOGY

This thesis has six phases and each one of them will be discussed in a separated chapter:

a. **Phase 1:** Building a dictionary for brute-force attacks. Dictionary is built by a script up to combination of possible digits, notations and characters in order to include all possible passwords can be entered by users. The work done in this phase will be discussed in chapter 3.
b. **Phase 2:** Developing software to be able to do brute-force attack with single CPU and multiple CPUs by using generated dictionary in Phase 1. The work done in this phase will be discussed in chapter 3.
c. **Phase 3:** Adding feature to the software to be able to do brute-force attack with single GPGPU and multiple GPGPUs by using generated dictionary in Phase 1. The work done in this phase will be discussed in chapter 3.
d. **Phase 4:** Adding feature to the software to be able to do brute-force attack with all GPGPUs on the network as distributed by using generated dictionary in Phase 1. The work done in this phase will be discussed in chapter 3.
e. **Phase 5:** Testing the software with generated dictionary for all methods and collect results.
f. **Phase 6:** Analyzing and comparing test results.

1.3. RESEARCH CONTRIBUTION

The primary contributions of the research work done in this thesis are described as follows:

a. A dictionary file including all possible passwords which can be used for SQLCipher encryption.
b. Software analysis, design and code examples for different methods on brute-force attacks such as CPU, GPGPU and distributed.

c. A comparison and discussion of SQLCipher brute-force attacks with different methods.

1.4. THESIS STRUCTURE

The remaining chapters of the thesis are constructed as follows:

a. **Chapter 2:** Presents a theoretical background of previous work related to the research work done in this thesis. The theoretical background covers two different identified subjects related to each objective of this thesis.
b. **Chapter 3:** Introduces generating dictionary file which includes all possible passwords for SQLCipher. Explaining which characters and why considered for dictionary generation and how much passwords required for SQLCipher. The chapter also mentions about software for all different methods, explains selected development environment, programming language and design of the software, gives code examples for each method and explains critical performance optimizations. Also graphical user interface capabilities is mentioned and some screenshots are given.
c. **Chapter 5:** Provides a discussion and summary of this thesis by reviewing the main objectives.
d. **Chapter 6:** Concludes the thesis by analyzing the main findings with respect to the thesis questions.

1.5. SUMMARY

This chapter has introduced the main outlines and briefs about the following chapters of this thesis. Research overview, methodology, and contributions were discussed. Research objectives and issues were discussed in this chapter.

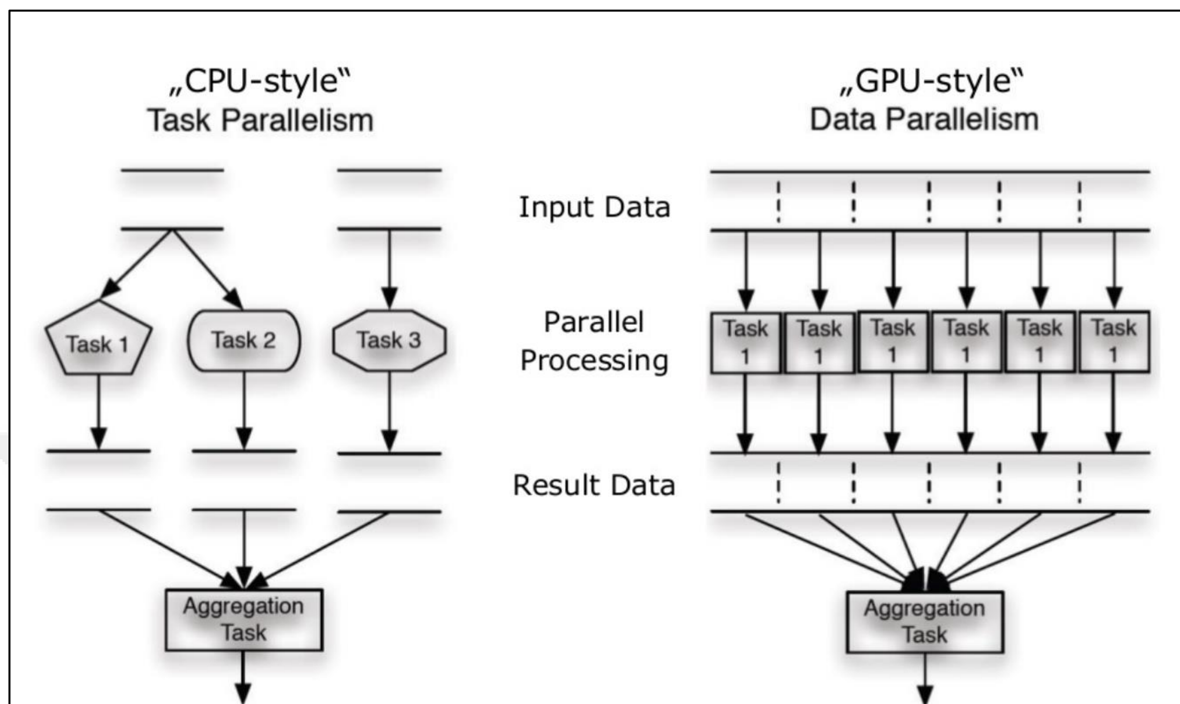
2. LITERATURE REVIEW

2.1. GPGPU

Graphical processing units are widely used in general purpose applications nowadays, especially in parallel and independent tasks execution applications. To be more specific, GPUs are widely used in the high performance computing applications including cryptography, data compression/decompression, image processing, AI, machine learning, crypto currency mining and blockchain technologies recently. And also like in our case, usually brute force attacker applications are targeting GPUs to be faster as Ahmadzadeh et al. (2017) mentioned, possible to get 20K+ performance even with remarkable less power consumption.

CPUs have limited core size but each core is capable of doing different huge tasks and they are for general purpose computing. CPUs can handle control flow intensive works efficiently. Because of the branching, memory latency is an important issue and caches are used for decreasing the latency. Besides, CPU cores may run totally different instructions and available for context switching which is fundamental requirement to operate general purpose. Unlike CPUs, GPUs have plenty of cores and all execute same instructions at the same time. GPU cores share the program counter and there is no branching, therefore there is no context switching.

Figure 2.1: CPU vs GPU Task Handling

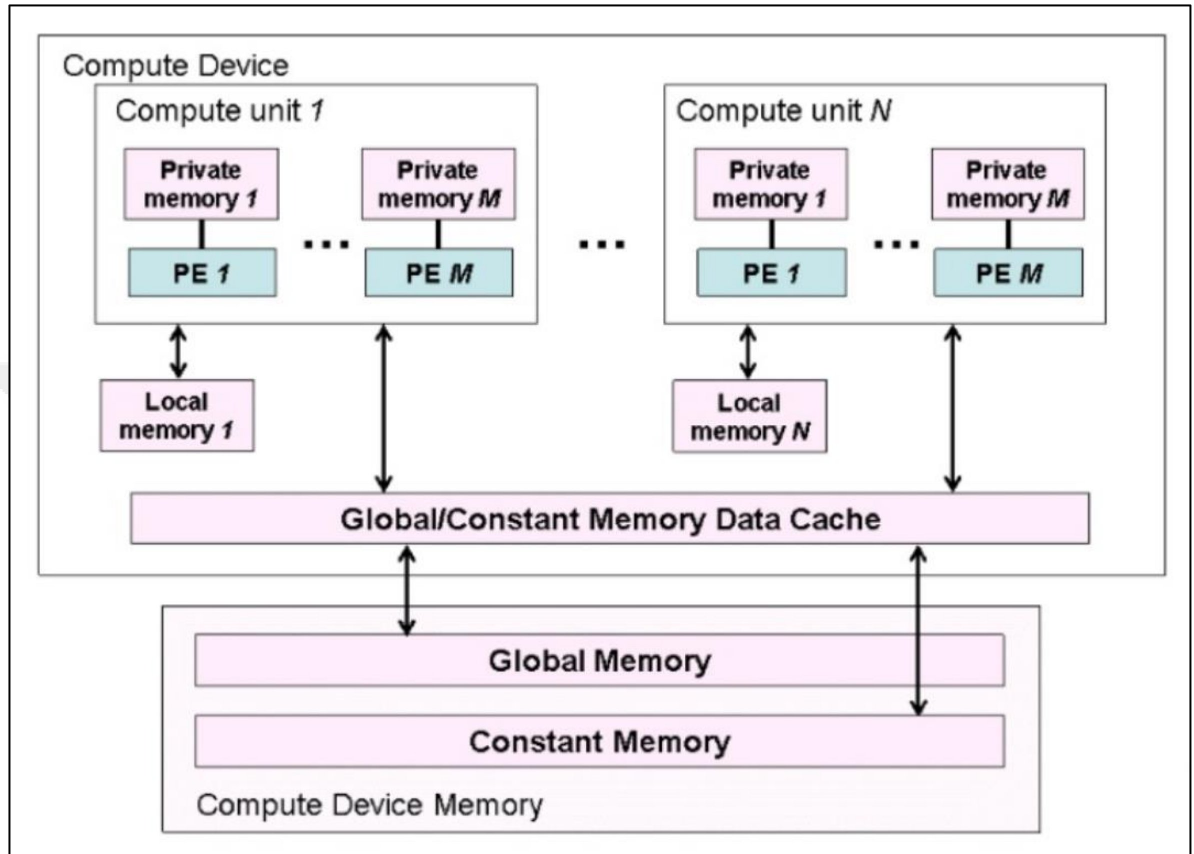


Source: Frank Feinbube, OpenHPI, Parallel Programming Concept, p.29

Cores are grouped small and bigger units which makes different levels in memory model of GPU architecture. Each core called work item and it has own private memory. Work items constitute a larger structure which called work group and local memory is accessible by all work group' work items and shared by them. Work groups are called *warp* in NVIDIA GPUs and *wavefront* in AMD GPUs. Work groups create compute units and global memory shared by all compute units which means accessible by every work item in GPU. Global memory is bigger than local memory and local memory is bigger than private memory. Private memory is closest to the core like register in CPU architecture and of course it is the fastest memory accessed by core.

Another important concept is "kernel". Kernels are the code executed in GPU work item and each work item has its own instance of running kernel. An ideal kernel for a GPU should has thousands of independent pieces of work which causes using all available compute units and allows context switching to hide latency.

Figure 2.2: GPU Memory Model



Source: Frank Feinbube, OpenHPI, Parallel Programming Concept, p.56.

2.2. OPENCL

OpenCL is a library interface for developing applications that may run on a large number of different platforms including consisting of CPUs, GPUs, DSPs, and FPGAs. OpenCL specifies the interface for controlling and use compute device for application programmers. It is a standard for device vendors to develop their drivers and middleware in order to make their device being used easily by any application developed up to OpenCL interface, hence provides platform independency. OpenCL is an open source standard shaped, managed and maintained by a group of vendors created a non-profit

consortium called Khronos group including ARM, AMD, IBM, Nvidia, Qualcomm, Xilinx etc.

OpenCL has a relaxed consistency block-based parallelism model runs on SMP, multithreaded and SIMD.

OpenCL let programmers to develop hybrid implementation of computationally intensive operations. Hofmann et al. (2018) used OpenCL for their cutoff-based particle interactions within the ScaFaCoS library test. They wanted to use both GPUs and multi core CPUs, so that OpenCL was chosen and observed that even for multicore CPUs there is still limited performance improvements.

As explained in by Hofmann et al. (2018);

> *The parallelization was implemented with OpenCL such that it can be executed with GPUs as well as multicore CPUs. The behavior of the OpenCL-based parallelization was analyzed for uniform and nonuniform particle distributions. In comparison to sequential executions with a CPU, significant reductions of the runtime were achieved with various GPUs. The usage of OpenCL for multicore CPUs showed only limited performance improvements due to remaining sequential parts and a high overhead.*

The biggest GPU vendors are AMD and Nvidia. AMD produce almost all devices with OpenCL support and widely used by OpenCL programmers. However, Nvidia has its own device interface library which called Cuda. Cuda is not open source, it is maintained by Nvidia and just for Nvidia devices. Recently, Nvidia has started to support OpenCL as well on their devices but since Cuda is specified for Nvidia chipsets, OpenCL could not give the exactly same performance on Nvidia devices yet. Also while OpenCL makes hybrid applications available, Cude is just for Nvidia GPUs, could not being execute on CPUs or other acceralater devices.

2.3. AES

The Advanced Encryption Standard(AES), also known as *Rijndael*, specifies a encryption/decryption algorithm to protect data. Algorithm specifications established by

NIST in 2001. It is a symmetric algorithm which means same key used for encryption and decryption. AES known as block cipher means working with block of data. AES uses three different length of keys which are 128, 192 and 256 bits for encryption and decryption of data on 128 bits block of data.

There are too many research about making AES faster and lately researches are tend to use GPGPUs instead of CPUs(Rosenband et al.(2009), Ma et al. (2017), Wang et al. (2019)). Mostly researches are concentrate on specific block cipher mode in order to benchmark the GPU performance for AES implementations. For instance Ma et al. (2017) worked on Cipher Block Chaining (CBC) mode and Wang et al. (2019) worked on Electronic Code Book (ECB) mode. As you can see from the figure below they both agreed on that if input data size is smaller than 4KB then CPU is faster than GPU. If input data size increase then GPU getting better up to 112 times faster than CPU.

Figure 2.3: CPU vs GPU AES Performance Up To Data Size

File Size (bytes)	Decryption			
	Time (seconds)		Throughput (bytes per second)	
	CPU	GPU	CPU	GPU
1202	0.000264	0.000771	4560606.06	1561608.30
4652	0.001168	0.000821	3993150.68	5680876.98
9302	0.002081	0.000790	4472849.59	11782278.48
18602	0.003889	0.000821	4785806.12	22669914.74
37202	0.007196	0.000918	5170094.49	40527233.12
74402	0.014296	0.001040	5204532.74	71542307.69
148802	0.028493	0.001331	5222475.69	111798647.63
297602	0.056839	0.002018	5235911.96	147474727.45
595202	0.113610	0.003577	5239010.65	166397539.84
1190402	0.227203	0.005847	5239385.04	203592269.54

Source: Canhui Wang, Xiaowen Chu, Hong Kong Baptist University, GPU Accelerated AES Algorithm, 2019

CBC mode mostly used in application today and SQLCipher use this mode as well. We work on first 1K data, so our input data size to AES algorithm is 1K which is smaller

than 4K. This means we should use CPU for decryption after we get key from PBKDF2 on GPU. This is why our solution is hybrid, including both CPU and GPU.

2.4. PBKDF2

One of the main weakness of many security systems is the strength of the chosen password (Abbas et al. 2014). If the password is not long enough and/or is not hard enough it would be easy for brute-force attacks. PBKDF2 aims to make passwords strength enough to slow down brute-force attacks. It applies pseudorandom function like HMAC to the password and repeat the process many times to get a final derived key which is more strength and long enough to be used as cryptographic key. It is not an encryption/decryption method, it is just for making password cracking more difficult since it requires computational power and also time for each password test. For instance, SqlCipher version 3 needs 64000 iterations on actual key to generate key to be used in encryption/decryption operation. Therefore, we can say that weakness of many security systems is not only strength of the chosen password but also password key derivation function used (Abbas et al. 2014).

PBKDF2 is well-known password-based key derivation function and it is very CPU-intensive operation. Recently researchers works on other accelerator devices, especially on FPGAs, to make operations faster (Cao et al. 2015, Visconti et al. 2018) and also more energy efficient way (Li et al. 2016).

In Figure below you can see the test result for PBKDF2 with HMAC in different setup. Using FPGA is more effective in speed and energy consumption since it is not a software, it consists of circuits. By the way, GPU performance also remarkable.

Figure 2.4: PBKDF2 Performance CPU vs GPU vs FPGA

Architecture	RIVYERA S6-LX150 (128 FPGAs)	4× Tesla C2070 GPUs [13]	Spartan6-LX150 (single FPGA)	Core i7-970 @ 3.2GHz (6 cores / 12 threads)
Passwords/sec	245,397	72,786	1,917	740
Speedup (vs. PC)	331.61	98.35	2.59	1
Energy consumption	937W	952W[1]	-	225W
Energy per 10^9 passwords	1.1kWh	3.6kWh	-	84.4kWh

[1] from the specifications of four Tesla C2070 GPUs

2.5. SQLCIPHER

SQLCipher is provides 256 bits AES encryption (CBC mode) for SQLite database files. It is an open source project and maintained by Zetetic LLC. has both community and commercial versions and can be used any platform including mobile devices.

SQLCipher encrypt and decrypt file block by block and the default block size for version 3 is 1024 bytes (4096 bytes in latest version 4). Each block has its own IV located at the end. SQLCipher use PBKDF2-HMAC-SHA512 function for key derivation. Iteration count is 64000 for version 3 (256000 in latest version 4).

SQLCipher is totally compatible with SQLite library, so any SQLite command/query also works with SQLCipher. It is lightweight and simple, so that prefered especially on small and mobile devices.

Well known applications using SQLCipher:

a. Signal Private Messenger
b. WICKR
c. ChatSecure(IOS)
d. Threema
e. Snapchat
f. WeChat

g. Wickr
h. Lulubox
i. IMO
j. BBM
k. KakaoTalk
l. Eagle

2.6. SUMMARY

In this chapter, we gave a brief theoretical background of brute-force field. The main methods and approaches were discussed as well. Background of the previous work and applications in brute-force on SQLCipher was explained.

3. DATA AND METHOD

We will follow some steps to get our results and evaluate the brute-force possibilities on SqlCipher. First we will generate big enough dictionary file to use in our tests. To do this we will write a Python script. After dictionary generation we will develop an application to do brute-force attack tests with different methods. We will use two type of GPU to make our test.

a. AMD Radeon RX 460
b. AMD Radeon RX 580

We prefer working on AMD and using OpenCL library while developing application since we will also be able to use the same code with NVidia type GPUs which supports both CUDA and OpenCL. Development will be done in C++ programming language on Linux environment.

3.1. DICTINARY CREATION

As we mentioned, Sqlcipher uses AES-256 for encryption which obviously means the encryption key size 256 bits. Key may be any combination of bits but since we almost always use keyboard as input device for password entering, we can ignore bits combination which does not entered from general used keyboards.

Usually passwords are composed of numbers and letters. If it is only numbers then the potential character pool size is 10 from 0 to 9. If the password is only letters, then the character pool size is 26 from A to Z, however if we consider letters incasesensitive then for both upper and lower case letters pool size is 52 instead of 26. If the password is alphanumeric, the character pool size will be 62 for digits and letters, each group is additive. If we think about for keyboard entered characters, such as #, $, %, the underscore character and others, then character pool size will be 95 (add 33 more characters) to the possible pool on most keyboards.

Table 3.1: Possible Character Set For Passwords

space	*	4	>	H	R	\	f	p	z
!	+	5	?	I	S	]	g	q	{
"	,	6	@	J	T	^	h	r	|
#	-	7	A	K	U	_	i	s	}
$	.	8	B	L	V	`	j	t	~
%	/	9	C	M	W	a	k	u	
&	0	:	D	N	X	b	l	v	
'	1	;	E	O	Y	c	m	w	
(	2	<	F	P	Z	d	n	x	
)	3	=	G	Q	[	e	o	y	

Passwords may be 1 character at least and 32 characters at most to be 256 bits key. If password is less than 32 characters then the remain bits will be 0 to make it 256 bits. Usually people use at most 8 characters password. So we may say possible password pool will consist of below pools:

1 character pool, size is 95

2 characters pool, size 95^2 (9025)

3 characters pool, size 95^3 (857375)

4 characters pool, size 95^4 (81450625)

5 characters pool, size 95^5 (7737809375)

6 characters pool, size 95^6 (735091890625)

7 characters pool, size 95^7 (69833729609375)

8 characters pool, size 95^8 (6634204312890625)

For finding size total possible password pool size we should add all results above and we will get below:

6704780954517120 -> ~ 6.7 x 10^15

Each character is one byte length, lets calculate how big file we will get If we generate all those possible passwords:

1 character pool, size is 95x1 (95 byte)

2 characters pool, size 9025x2 (18050 byte)

3 characters pool, size 857375x3 (2572125 byte ~ 2.5 MB)

4 characters pool, size 81450625x4 (325802500 byte ~ 310 MB)

5 characters pool, size 7737809375x5 (38689046875 byte ~ 36 GB)

6 characters pool, size 735091890625x6 (4410551343750 byte ~ 4107 GB)

7 characters pool, size 69833729609375x7 (488836107265625 byte ~ 455264 GB)

8 characters pool, size 6634204312890625x8 (53073634503125000 byte ~ 49428673 GB)

If we add all then we get about 50 peta bytes (50 x 10^15) which is very big data even to store.

Of course, applications may determine password policy in different way. For our test we will try to narrow possible password set to be able to generate and use. Our password policy will be as below:

a. Alphanumeric
b. Case insensitive
c. At most 6 characters
d. Only '_', ';', '-', '#', '$', '%', '*', '?', '=', '!' and ',' special characters are allowed

Since passwords are case insensitive only 28 letters will be considered. Additionally, there are 10 digits and there is only 11 special characters. So there are 49 characters in character pool. We will generate passwords at most 6 characters passwords, lets calculate the possible password size up to our policy:

1 character pool, size is 49

2 characters pool, size 49^2 (2401 items and 4802 bytes)

3 characters pool, size 49^3 (117649 items and 352947 bytes)

4 characters pool, size 49^4 (5764801 items and ~ 22 MB)

5 characters pool, size 49^5 (282475249 items and ~ 1.3 GB)

6 characters pool, size 49^6 (13841287201 items and ~77.3 GB)

If we add all we will get about 14 billion item and about 80 GB data. We will write a python code to generate test dictionary.

3.2. DICTINARY LOAD METHODS

Usually dictionaries used in brute-force are very huge to load all into memory. Operating System will Even if we choose some assumption for our dictionary keys to reduce the size of dictionary file or loaded key, still there will be a huge dictionary file. Today's computers or servers usually use memory between 8GB and 128GB, however our dictionary is about 200GB. So we need to consider dictionary files bigger than machine's memory size.

So that, at the beginning, application checks the dictionary file size (isSmallDictionary()), if size is less than 200MB then reads all file into memory line by line (runSmallDictionarySizeMode()). 200MB is very small size and affordable for today's computers. If the size is bigger than 200MB then dictionary processor thread reads file part by part to keep memory usage lower (runBigDictionarySizeMode()).

Table 3.2: Dictionary Processor Header

```
#pragma once
#include <memory>
#include <thread>
#include <fstream>
#include "Parameters.h"
#include <vector>

class DictionaryProcessor
{
public:
    explicit DictionaryProcessor(int microseconds = 100, int workPerIteration = 200);
    ~DictionaryProcessor();
    std::thread& getWorkerThread() { return m_workerThread; }
    bool init(const Parameters& params);
    bool isSmallDictionary();
    void start();
    void stop();
    void waitToFinish();
    static bool isReadFinished;

private:
    /* thread related */
    void run();
    void runSmallDictionarySizeMode();
    void runBigDictionarySizeMode();

    bool m_isThreadRunning;
    int m_microseconds;
    int m_workPerIteration;
    std::ifstream m_dictionaryFile;
    Parameters m_parameters;
    std::thread m_workerThread;
};
```

3.3. SQLITE FILE VALIDATION

If we use SQLCipher library then we do not need to validate decrypted data if it is valid sqlite file data or not. However we do not want to use SQLCipher library so we should do decrypted data validation in order to understand decrypted data is sqlite file data or not. In fact, it is very crucial operation since we determine if password correct or not up to decrypted data validation.

Like all other file formats, Sqlite File also has a format. We can check decrypted data if fits Sqlite File header format because we use file first 1024 bytes which includes file header. Sqlite file has some mandatory fields in header and we can check those fields to determine validation.

Table 3.3: SQLite File Checker Data

```
struct SQLiteHeaderData
{
    short pageSizeInBytes;
    char fileFormatWriteVersion;
    char fileFormatReadVersion;
    char ignore1;
    char maxEmbeddedPayloadFraction;
    char minEmbeddedPayloadFraction;
    char leafPayloadFraction;
    char ignore2[20];
    unsigned int schemaFormatNumber;
    char ignore3[8];
    unsigned int textEncoding;
} __attribute__((packed));
```

As you may see from above in Table 3.3, there are some fields application read from file and gotten to compare with SQLite file standard header values. Below datas are enough to figure file out weather is it a valid SQLite file or not.

a. File format write version. 1 for legacy; 2 for WAL.
b. File format read version. 1 for legacy; 2 for WAL.
c. Maximum embedded payload fraction. Must be 64.
d. Minimum embedded payload fraction. Must be 32.
e. Leaf payload fraction. Must be 32.
f. The schema format number. Supported schema formats are 1, 2, 3, and 4.
g. The database text encoding. 1 for UTF-8, 2 for UTF-16le, 3 for UTF-16be.

In Table 3.4 below method is verifier for data if it is SQLite file or not by checking the list above.

Table 3.4: SQLite File Checker Method

```
bool BufferDecryptor::isSQLiteFile(const char* firstPageWithoutStringHeader)
{
   struct SQLiteHeaderData* gHeaderData = (struct
SQLiteHeaderData*)firstPageWithoutStringHeader;
   auto writeVersion = int(gHeaderData->fileFormatWriteVersion);

   if (writeVersion != 1 && writeVersion != 2)
      return false;

   auto readVersion = int(gHeaderData->fileFormatReadVersion);
   if (readVersion != 1 && readVersion != 2)
      return false;
   if (int(gHeaderData->maxEmbeddedPayloadFraction) != 64)
      return false;
   if (int(gHeaderData->minEmbeddedPayloadFraction) != 32)
      return false;
   if (int(gHeaderData->leafPayloadFraction) != 32)
      return false;

   unsigned int format = gHeaderData->schemaFormatNumber;
   unsigned int leFormat = be32toh(gHeaderData->schemaFormatNumber);
   if ((format != 1 && format != 2 && format != 3 && format != 4) &&
         (leFormat != 1 && leFormat != 2 && leFormat != 3 && leFormat != 4))
      return false;

   unsigned int encoding = gHeaderData->textEncoding;
   unsigned int leEncoding = be32toh(gHeaderData->textEncoding);
   if ((encoding != 1 && encoding != 2 && encoding != 3) &&
         (leEncoding != 1 && leEncoding != 2 && leEncoding != 3))
      return false;

   return true;
}
```

3.4. CPU BASED METHOD

CPUs are general purpose computing units and used for all kind of operations. They are not specified for any type of operations. It is good to work on CPUs since we can do anything and all computers have CPUs. In this part we will try to use only CPU devices to make brute-force attack by using single or multi CPUs.

Our test machine has 8 core AMD processor. Some features:

CPU MHz: 2212.223
CPU max MHz: 3800,0000
CPU min MHz: 1400,0000
BogoMIPS: 7635.84
L1d cache: 16K
L1i cache: 64K
L2 cache: 2048K
L3 cache: 8192K

3.4.1. Single CPU with SqlCipher Library

Sqlcipher provides a library to access, read and write data from encrypted file. With sqlcipher API you can create an encrypted file with password. Sqlcipher takes given password and does required operation in library, so you just need to use API and no need to touch any encryption/decryption process.
In our application we will use single CPU core and hence single thread to find sqlcipher encrypted database file password. In order to do this first we will read dictionary file to get passwords and use Sqlcipher library methods to find the password of the encrypted file database. Application will read given dictionary part by part because of the memory limitation for big dictionary files and try to test taken passwords by using Sqlcipher library. Regarding testing the passwords, we will use two methods in Sqlcipher library to figure out if password is true or not. We will use below two methods from API:

sqlite3_key(): Tell sqlcipher to use which password for encryption/decryption

Table 3.5: SQLite Password Test Method Decleration

```
/*
** Specify the key for an encrypted database.  This routine should be called right after
sqlite3_open().
** The code to implement this API is not available in the public release of SQLite.
*/
SQLITE_API int sqlite3_key (
  sqlite3 *db,                  /* Database to be rekeyed */
  const void *pKey, int nKey     /* The key */
);
SQLITE_API int sqlite3_key_v2 (
  sqlite3 *db,                  /* Database to be rekeyed */
  const char *zDbName,          /* Name of the database */
  const void *pKey, int nKey     /* The key */
);
```

sqlite3_exec(): Execute an sql query on database file and get the results

Table 3.6: SQLite Query Execution Method

```
SQLITE_API int sqlite3_exec (
        sqlite3*,                              /* An open database */
        const char *sql,                       /* SQL to be evaluated */
        int (*callback)(void*,int,char**,char**),  /* Callback function */
        void *,                                /* 1st argument to callback */
        char **errmsg                          /* Error msg written here */
);
```

As you can see both two methods require db file pointer, so initially application open the database file. Our test method takes each read password from dictionary file one by one and tell sqlcipher to set this password as the key for operations. After that application calls a simple generic query and try to get result. If the set key is not true then method throws an exception, so application can understand password is not correct.

Table 3.7: Password Processor Worker Method

```
void PasswordProcessor::workerFunction()
{
    try {
        sqlite3_key(m_pSqliteDb, passwd.c_str(), passwd.length());
        if (sqlite3_exec(m_pSqliteDb, "SELECT count(*) FROM sqlite_master;", nullptr,
nullptr, nullptr) == SQLITE_OK) {
            m_password = passwd;
            m_found = true;
            m_isThreadRunning = false;
        }
    } catch (std::exception& e) {
    }
}
```

In sqlite3_key() function, encrypted sqlite file pointer is taken with password to be used in decryption. Function calls SqlCipher library in order to do all operations including password based iteration (64000 iterations), reading 1024 bytes of file and finally decryption attempt using generated last key. All done by library by using CPU and if password is not true then exception is thrown.

In our test machine with one core, application is able to test 5 passwords in second. In Figure 3.1 below you can see only one CPU core is used by application. There are two threads are shown in htop screen but one is main thread the other is worker thread which does actual job.

Figure 3.1: Single CPU Usage Test Threads Showing on Htop

3.4.2. Multi CPU with SqlCipher Library

As we mentioned in the previous section, one of the method for password checking is using SQLCipher library. We developed and tested with one CPU. In this section we will develop and test with all CPUs on system.

For using all CPUs we will create one thread for each CPU. If CPU has hyperthreading support then we may make thread count double. Below you may see there is a vector for keeping threads. For each CPU/core, application create a thread and push into thread vector. After all threads created then application iterate over thread vector and start each thread. In order to wait for threads, each thread joined.

Table 3.8: Multi CPU Usage Threads Join

```
int multi_thread(DictionaryProcessor& dp, Parameters& p)
{
   auto coresCount = std::thread::hardware_concurrency();
   LOG_INFO << "USED_CORES: " << coresCount << std::endl;

   std::vector<std::shared_ptr<PasswordProcessor>> ppVector;
   for (auto i = 0; i < coresCount; i++) {
      auto ppPtr = std::make_shared<PasswordProcessor>(dp.getPasswordQueue());
      if (!ppPtr->init(p)) {
         LOG_ERROR << "PASSWORD_PROCESSOR_INIT_FAILURE" << std::endl;
         return EXIT_FAILURE;
      }
      ppVector.push_back(ppPtr);
   }

   dp.start();
   dp.getWorkerThread().join();

   std::for_each(ppVector.begin(), ppVector.end(), [](auto ptr) { ptr->start(); });
   std::for_each(ppVector.begin(), ppVector.end(), [](auto ptr) { ptr->
getWorkerThread().join(); });

   // after threads finished
   if (!PasswordProcessor::m_found) {
      std::cout << "PASSWORD_NOT_FOUND" << std::endl;
      return EXIT_FAILURE;
   }

   std::cout << "PASSWORD: " << PasswordProcessor::m_password << std::endl;
   return EXIT_SUCCESS;
}
```

In our test machine, with using all 8 CPUs -means 8 threads- application is able to test about 38 passwords in second which is an expected performance.

In Figure 3.2 below we can see all CPU cores are used by application because of multi threads.

Figure 3.2: Multi CPU Usage Test Threads Showing on Htop

3.4.3. Single CPU with SSL Library and Buffered Data

SQLCipher library does everything by itself, however we can do it by using SSL library. This will have some advantages and disadvantages as well. Main advantage is we do not need to read file from filesystem, instead we can read first 1024 bytes of file to memory and work with buffer area. Reading file from filesystem/disk is very slow, on the other hand reading memory is very fast. Disadvantage of this method is we need to validate decrypted data.

Let's follow the steps what application should do in this method:

a. First, we need to do key iteration. For this, we use SSL library PBKDF2 function. This function will take password, password salt data, size of data and key, size of key and iteration count as argument

b. At this point, our key iteration done and key for decryption is ready. We will use SSL decryption function to decrypt file first 1024 byte read to memory.

c. Last step is check if decryption output is valid Sqlite file. Sqlite file format has some mandatory values as file header. We check those values and sure about file has valid sqlite format or not. If format is valid then we can be sure about that the password is correct and decryption is success.

Table 3.9: Iteration and Decryption with SSL Library Method

```
bool BufferDecryptor::checkPassword(const std::string &passwd)
{
   const char* pass = passwd.c_str();
   PKCS5_PBKDF2_HMAC_SHA1(pass, strlen(pass), m_passwdSalt,
m_kSqliteHeaderSize, m_kPbkdfIterationCount, keySize, m_key);

   auto outPtr = m_outputBuffer;
   EVP_CIPHER_CTX ectx;
   EVP_CipherInit(&ectx, m_evpCipher, m_key, m_iv, 0);
   EVP_CIPHER_CTX_set_padding(&ectx, 0);
   EVP_CipherUpdate(&ectx, outPtr, &tmp_csz, m_inputBuffer + m_kSqliteHeaderSize,
         m_kSqlitePageSize - reserveSize - m_kSqliteHeaderSize);
   outPtr += tmp_csz;
   EVP_CipherFinal_ex(&ectx, outPtr, &tmp_csz);
   EVP_CIPHER_CTX_cleanup(&ectx);

   return isSQLiteFile((const char*)m_outputBuffer);
}
```

In test, a small dictionary file which has 727 lines finished in 46.66 seconds by a test machine using only a single core. Performance is about 15 passwords per second which better than using SQLCipher library more than twice.

3.4.4. Multi CPU with SSL Library and Buffered Data

In previous section we saw single CPU test result with SSL library is more better than single CPU test with SQLCipher library. Since we use all cores with SSL library of course we will get better than SQLCipher library using all cores.

In test, dictionary file has 38650 lines and finished in 470.05 seconds in our test machine by using all 8 cores.

38650 / 470.05 / 8 = 10.27

Performance is about 10 passwords per second which is smaller than single CPU with SSL library test. In single core test there are other 7 cores for operating system to use, so that there are not too much context switch for out thread. On the other hand, in multi cpu test, application try to use all CPU cores but it is not possible to use all CPU cores without too much interruption because operating system should use CPU cores. So there are too much context switch for threads.

3.5. GPU BASED METHODS

In previous section we tried to do brute-force attack by using CPUs. The performance is related to CPU cores, if we are able to increase core count then we will get more performance up to count. However it is not that chip to use too many CPU cores and also there are some constraints on count of CPU cores in a single chip. So we need to use multiple servers which has for instance 64 CPU cores to get better and somehow acceptable performance.

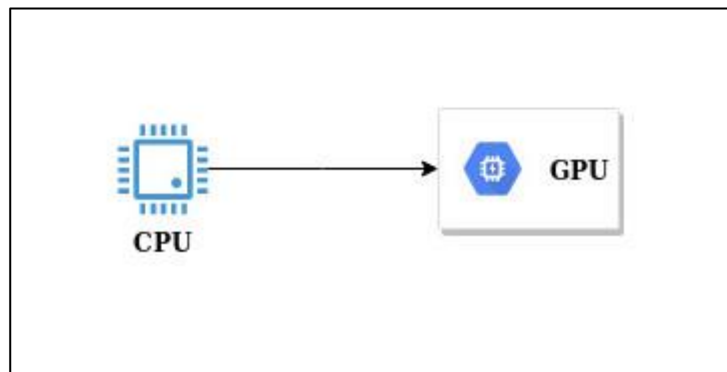
On the other hand, GPUs have too many cores, and can be used for general computing as well. Using GPUs is very efficient and chipper for parallelizable and independent works. Brute-force attacks like our test are also suitable for GPU using.

We will use two different GPUs in our tests. For single GPU test we will use AMD Radeon RX 580 which has 2304 cores (work item), 36 units (work group) and each unit has 64 cores. This means we can do 2304 process together in parallel by using GPU. For multiple and distributed GPU test scenarios we will use AMD Radeon RX 460, which has 896 cores.

3.5.1. Single CPU and Single GPU with Buffered Data

In this section we will use single CPU and single GPU. We will read all dictionary and calculate chunk size of passwords to be checked by GPU cores in parallel.

Figure 3.3: Single GPU Controlled With Single CPU



Let's follow the steps what application should do in this method:

a. First, we will read dictionary; if it is small enough (<200MB) we will read all in once, otherwise part by part to use memory properly. This is done by using CPU.

b. Second, we will read first 1024 bytes of encrypted file, as in Buffer Data method. This is done by using CPU.

c. Then, we need to do key iteration. For this, we will use GPU to make it parallel. We have implemented OpenCL PBKDF2 kernel/function. This function will take four arguments:

    i. password array which will be used for iteration
    ii. output array which will keep iterated result keys
    iii. password salt data which is part of encrypted file which read at the start of application
    iv. iteration count which is 64000

d. After we get iteration results for chunk of passwords, we will check each key in a loop if it is true key for decryption. This is done by using CPU.

    i. After decryption still we need to validate decrypted data by checking if it is valid Sqlite db file header or not

Now we will explain how the application interact with the GPUs step by step. In order to make GPU to do tasks we should give inputs and get the outputs. For this reason in the Table 3.8 you will see the preparation of input buffer array and output buffer array allocations. Passwords to be checked are copied into the input array buffer to be sent

GPU for PBKDF2 iterations. The size of arrays is password list size which is the size of GPU cores we have in working GPU.

Table 3.10: Input/Output Buffer Preperation for GPU Kernel

```
stInputBuffer* inbufArray = new stInputBuffer[passwdListSize];
stOutputBuffer* outbufArray = new stOutputBuffer[passwdListSize];

std::memset(inbufArray, 0, passwdListSize*sizeof(stInputBuffer));
std::memset(outbufArray, 0, passwdListSize*sizeof(stOutputBuffer));

for (int i=0; i<passwdList.size(); i++) {
        strToIntegerArray(passwdList[i].c_str(), passwdList[i].length(),
                    inbufArray[i].buffer, sizeof(inbufArray[i].buffer));
        inbufArray[i].length = passwdList[i].length();
}
```

Those input and output arrays are ready in host memory, however GPU cores can not use host memory, so we need to allocate memory in GPU memory are by copying our inputs as well. In the table 3.8 you see three arrays are created with different options. Input array is created read only by filling with input buffer of host which has passwords to iterate. Salt data is also copied from host memory. Output buffer are in the GPU will be filled with iterations results, so it is write only which will be written by GPU cores.

Table 3.11: Copy Host Memory Buffer to GPU Memory

```
cl::Buffer inputBufArray_clmem(m_context, CL_MEM_READ_ONLY |
                               CL_MEM_COPY_HOST_PTR,
                               passwdListSize * sizeof(stInputBuffer),
                               inbufArray);

cl::Buffer saltArray_clmem(m_context, CL_MEM_READ_ONLY |
                           CL_MEM_COPY_HOST_PTR,
                           sizeof(stInputBuffer), m_saltArray);

cl::Buffer outputBufArray_clmem(m_context, CL_MEM_WRITE_ONLY,
                                passwdListSize * sizeof(stOutputBuffer));
```

GPU executed cores are called “kernel” in OpenCL, so we should share “kernel” with GPU and we will give buffer areas and iteration count as arguments as in table 3.9.

Table 3.12: Kernel Parameters

```
cl::Kernel kernel(m_program, "func_pbkdf2");
kernel.setArg(0, inputBufArray_clmem);
kernel.setArg(1, outputBufArray_clmem);
kernel.setArg(2, saltArray_clmem);
kernel.setArg(3, m_kPbkdfIterationCount);
```

Since GPU has three dimensions we should tell GPU how many cores it should use by setting dimensions. After setting dimension we should enqueue the kernel to be executed by GPU and wait reading the output after GPU finished iterations and filled the output memory. As you see table 3.10 we can wait in host code by using flush() and finish() methods of OpenCL command queue.

Table 3.13: Work Group/Item Counts

```
auto globalWorkItemSize = std::min(passwdListSize, m_kGlobalWorkItemSize);
auto localWorkItemSize = std::min(globalWorkItemSize, m_kLocalWorkItemSize);
cl::NDRange global(globalWorkItemSize);        // Max Work Item Size
cl::NDRange local(localWorkItemSize);   // Prefered Work Group Size
m_queue.enqueueNDRangeKernel(kernel, cl::NullRange,  global, local);
m_queue.enqueueReadBuffer(outputBufArray_clmem, CL_TRUE, 0,
                                   passwdListSize * sizeof(stOutputBuffer), outbufArray);

m_queue.flush();

m_queue.finish();
```

At this point we have all keys which derived from passwords given to GPU. The 99 percent of time consuming tasks are finished and it is time to check the generated keys if there is a valid key exist. In table 3.11 all keys are test by calling “checkPbkfd2Sha1ResultKey” method and if somehow there is valid password we will generate log and set the found password before breaking the loop.

Table 3.14: Test Derived Keys

```
std::string result = "";
for (int i=0; i<passwdListSize; i++) {
    unsigned char keyStr[32];
    std::memcpy(keyStr, outbufArray[i].buffer, 32);

    if (checkPbkfd2Sha1ResultKey(keyStr)) {
        LOG_CRITICAL << "PASSWORD FOUND: " << passwdList[i] << std::endl;
        result = passwdList[i];
        break;
    }
}
```

We use SSL library in this case for decryption. After decryption by given derived key, we will get binary data which might be SQLite file header data. We should check the mandatory fields of SQLite file header as described in “SQLite File Validation” section. This is done by calling “isSQLiteFile” method.

Table 3.15: Derived Key Test Method

```
bool BufferDecryptor::checkPbkfd2Sha1ResultKey(unsigned char* testKey)
{
    auto outPtr = m_outputBuffer;

    EVP_CIPHER_CTX ectx;
    EVP_CipherInit(&ectx, m_evpCipher, testKey, m_iv, 0);
    EVP_CIPHER_CTX_set_padding(&ectx, 0);
    EVP_CipherUpdate(&ectx, outPtr, &tmp_csz, m_inputBuffer + m_kSqliteHeaderSize,
            m_kSqlitePageSize - reserveSize - m_kSqliteHeaderSize);
    outPtr += tmp_csz;
    EVP_CipherFinal_ex(&ectx, outPtr, &tmp_csz);
    EVP_CIPHER_CTX_cleanup(&ectx);

    return isSQLiteFile((const char*)m_outputBuffer);
}
```

We used AMD Radeon RX 580 GPU, have tested with two dictionary files, one has 38650 lines and the bigger one has one million lines. In both file, the password is at the end, so all passwords have been tested. As you may see from the below Figure 3.4,

performance is very good. For small file, it takes 4.54 seconds to test 38650 passwords which means performance is 8500 passwords per second. For bigger file, it takes 77.49 seconds to test one million passwords which means performance is about 13000.
With one CPU core we were able to test about 15 passwords per second and now we could check 13000 password with one GPU which is extremely better.

Figure 3.4: Performance Test with Single Gpu Using Two Dictionaries

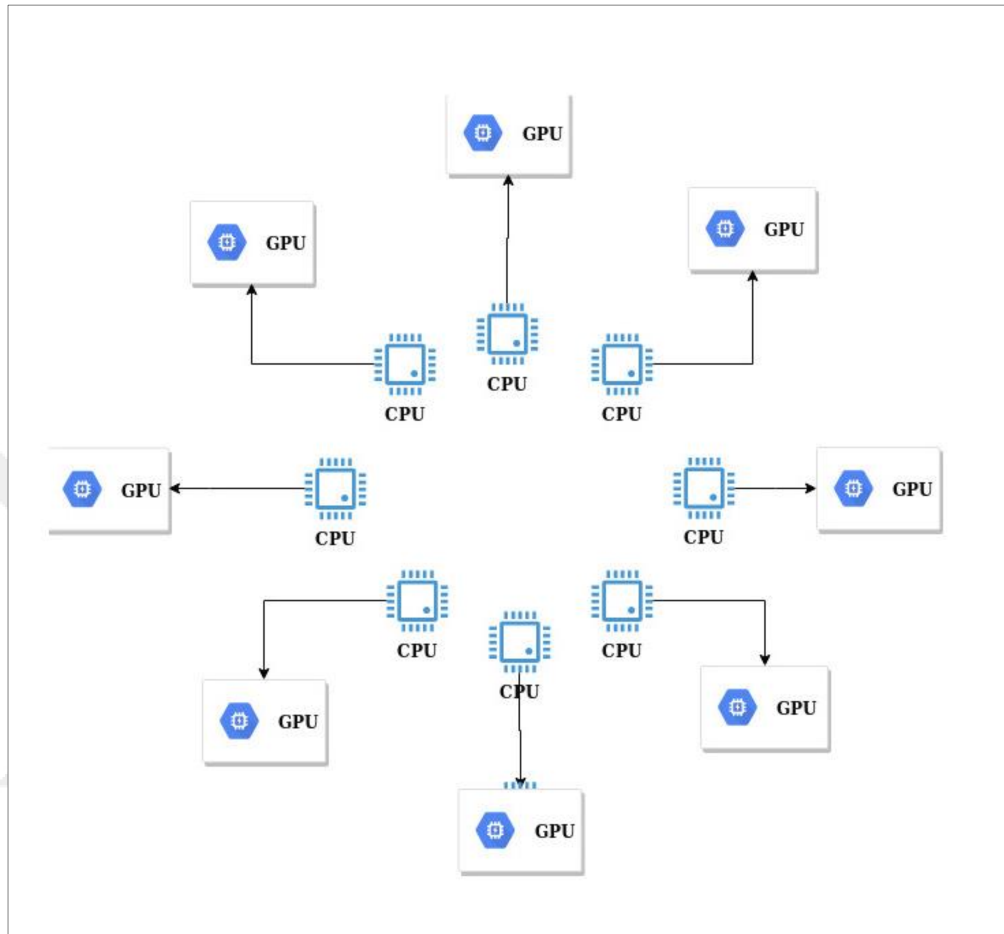
```
mgundes@shadow:$
mgundes@shadow:$
mgundes@shadow:$
mgundes@shadow:$ time ./sqliteintruder resources/38650-password-sktorrent.txt resources/enc_eagle.db
verbose: 0, skipCount: 0

real    0m4,543s
user    0m1,160s
sys     0m0,088s
mgundes@shadow:$
mgundes@shadow:$
mgundes@shadow:$
mgundes@shadow:$ time ./sqliteintruder resources/1000000-password-seclists.txt resources/enc_eagle.db
verbose: 0, skipCount: 0

real    1m17,494s
user    0m3,849s
sys     0m0,216s
mgundes@shadow:$
```

If we use RX 460 instead, then we are able to test 3600 passwords per second since RX 460 has less core count and memory. However it is 500 times better than single CPU usage.

3.5.2. Multi CPU and Multi GPU with Buffered Data

Since we get an extremely better result with single GPU, it is good to test multiple GPU connected to one machine. In this test scenario we have 8 AMD Radeon RX 460 GPUs and 2 CPUs.

Figure 3.5: Multi GPU Controlled with Multi CPU



AMD RX 460 has less core count and memory than RX580. In the previous test we got 3600 password test performance per second from RX 460 GPU. If we use 8 RX 480 GPU with a machine which has 2 CPUs, then we get about 26000 password test per second. It is almost 4000 times better than single CPU usage.

3.6. DISTRIBUTED METHODS

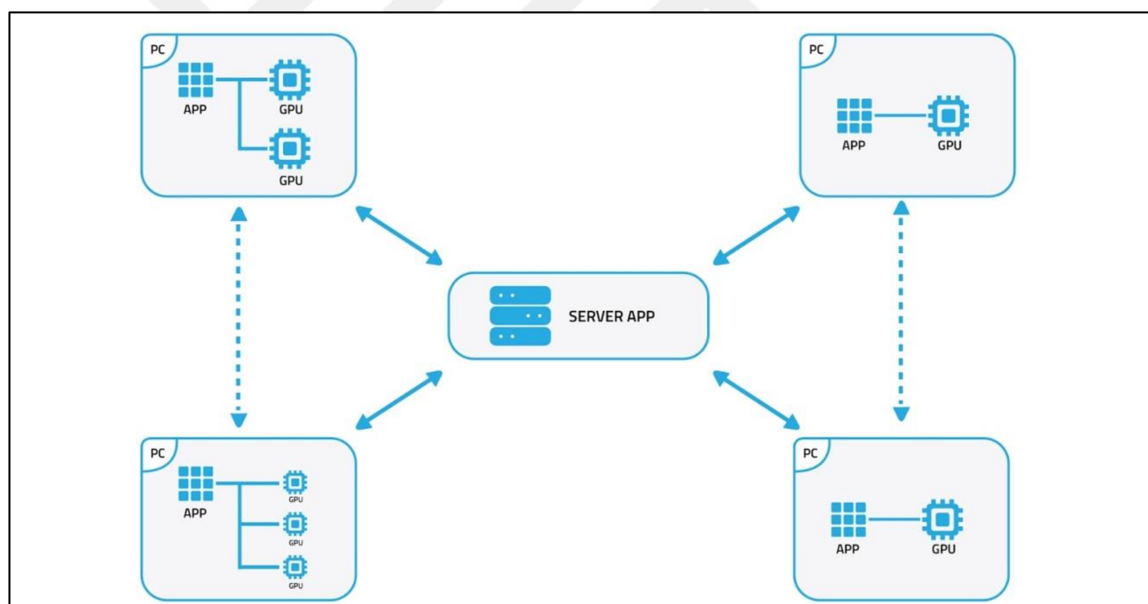
As we conducted from previous tests, GPUs are incredibly faster than CPUs for our brute-force attack scenarios. So that there is no need to develop and do test with distributed CPUs. We will add last feature to our application in order to do distributed

GPU test. Distributed means resource from different machines are used for same purpose over network. We will use all GPUs in the network to get more GPU power and test our dictionary as soon as possible.

3.6.1. Distributed GPUs

As shown in the below diagram, application will act as client and we will develop another application to act as a server. Server will read the encrypted file 1024 bytes data and dictionary. Dictionary will be read part by part and will be shared over network. Clients will get encrypted file data and passwords over network and share test results with Server.

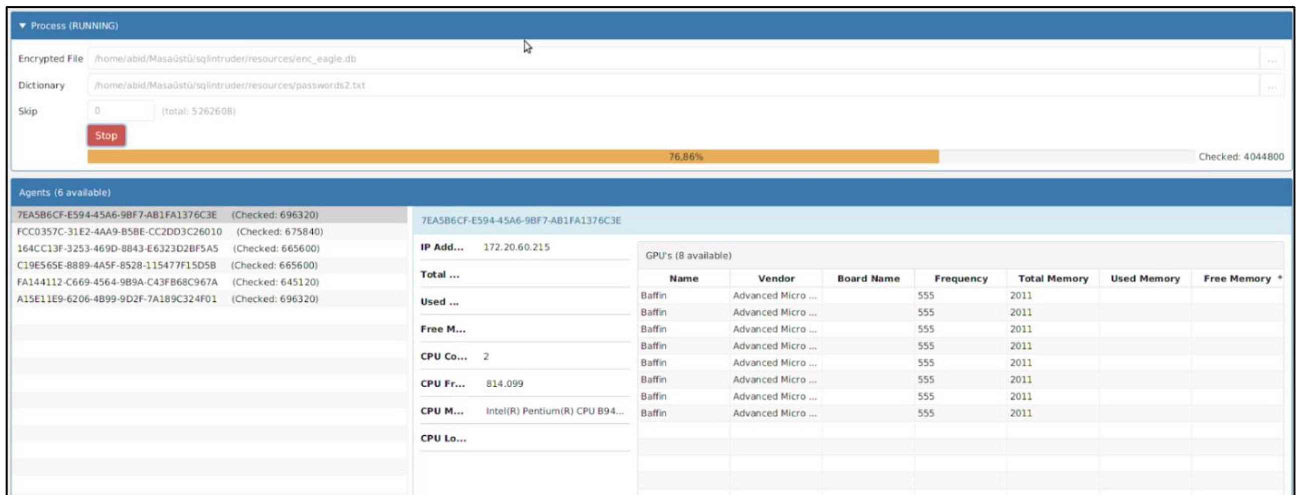
Figure 3.6: Distributed Single/Multiple GPUs on Different Machines



To make application distributed we will use Hazelcast in memory data grid open source library. Client is developed with C++ programming language, so we will use Hazelcast C++ client adapter library. On server side, we will develop a java application by using Hazelcast Java library.

The Figure 3.7 demonstrates server application UI. We can select dictionary and encrypted file from filesystem. We can see the running client applications and their informations like IP address, how many GPUs they have etc. Each client will have hazelcast UUID to distinguish them and also there is information about how many password is checked by that client next to the UUID.

Figure 3.7: Server UI, Testing with Six Machines Each Has 8 GPUs



We can start the client to test by clicking the Start button and the progress bar will be filled up to checked password percentage.

Below you can see the client side of dictionary processor main loop code. It waits for connecting server and after that it tries to get passwords from the distributed hazelcast password queue. For each chunk of passwords, it create a vector and push to global password vector in order to be accessible by password processor threads which responsible with testing passwords on GPUs. Since password test works as the previous there is no change on password processor class after application made distributed.

Table 3.16: Distributed Password Processor Main Loop

```
uint32_t totalTried = 0;
LOG_INFO << m_threadIndex << ". Thread starts to checking items.. ";
 m_bufferDecryptor.initialize(m_sqlcipherFileData, m_assignedGpuIndex);
 while (m_found == 0 && m_isThreadRunning != 0 &&
          Globals::IS_HAZELCAST_CLIENT_CONNECTED)
 {
    std::vector<std::string> dataVector;
    auto hasModeDataVector = Globals::DictionaryVector.pop_front(dataVector);
    if (!hasModeDataVector)
    {
       LOG_WARNING << m_threadIndex << ". There is not data vector at the moment.";
       std::this_thread::sleep_for(std::chrono::seconds(1));
       continue;
    }

    handleDataVector(dataVector);
    totalTried += dataVector.size();
    Globals::TOTAL_CHECKED_PASSWORDS += dataVector.size();
    LOG_INFO << m_threadIndex << ". Thread checked so far: " << totalTried;
    std::this_thread::sleep_for(std::chrono::microseconds(m_microseconds));
    LOG_DEBUG << "TOTAL CHECKED PASSWORDS BY HOST: "
                         <<  Globals::TOTAL_CHECKED_PASSWORDS;
    HazelcastDataHandler::GetInstance()->setTotalCheckedPassword(
                                                      Globals::TOTAL_CHECKED_PASSWORDS );
 }
```

If any password processor thread found the true password, again the result shared via hazelcast and lets the server application know about it. When server informed about password finding, then it tells all clients to stop and show password the server application user as in the Figure 3.8 below.

Figure 3.8: Server UI, Password Found

4. DISCUSSION AND CONCLUSION

Password cracking is very interesting topic for ordinary users, professionals and off course to the governors. There are many resources used for this issue. However it is not easy to crack a password by using brute-force attack since up to algorithms there are extremely more possibilities to test. The main criteria is time, if it is not feasible in time then solution could not be applied to any practical case.

However, it is not easy to figure out the weakness of an algorithm, hence most frequently the only way to crack a password is brute-force.

For SQLCipher, it uses AES256 for encryption and there is not any other way for password cracking except brute-force. We have tested seven different approaches for brute-force, four use CPUs and three use GPUs. The table below shows the performance for each approaches.

Table 4.1: Methods' performance results

Test Approach	Environment	Test Per Second Total	Test Per Second For Each
qlcipher Library with Encrypted File	Single CPU	5	5
Sqlcipher Library with Encrypted File	Multi CPU (8 Cores)	38	4.9
SSL Library with Buffer	Single CPU	15	15
SSL Library with Buffer	Multi CPU	80	10
SSL Library with Buffer + OpenCL	Single CPU + Single GPU (AMD RX 460)	3500	3500
SSL Library with Buffer + OpenCL	Single CPU + Single GPU (AMD RX 580)	8500	8500
SSL Library with Buffer + OpenCL	Multi CPU + Multi GPU (7 AMD RX 460)	23100	3300
SSL Library with Buffer + OpenCL	Distributed CPUs and GPUs (8 machines, 56 GPUs)	180000	3250

As can be seen from the table, password cracking for SQLCipher or any other application using long keys could not be done by using CPUs. Compare to accelerator devices CPUs are totally bad selection for brute-force attacks, at least for big dictionaries. FPGA devices are best choice if we only consider the speed of devices. However, it is not easy to use FPGAs in brute-force setups since FPGAs are hard to work with, compilations may take days while developing crackers. Also FPGA is not a chip solution. In addition it is not easy to extend your setup if you use FPGAs because it is not used by regular end users. On the other hand, GPUs are used widely and developing crackers for GPUs are not too hard. Also, since you have software in GPU setups, you can extend your setup easily even with regular users' GPUs. Since GPUs are run software, you can use same GPUs for different purposes only by using different software, no need to recompilation

as in FPGAs. Furthermore, GPU industry developing too fast and there are very powerful devices are reachable even by normal users. Hence, we are sure about that someone should use GPUs for password cracking if it is not possible with CPUs like in big dictionary cases or more.

Also from the table, we can say that getting 180K password test per second performance with a small setup is possible. In addition, there are very advanced GPU devices nowadays such as NVIDIA TeslaV100 which has more than 5000 cores and each core more faster than usual GPUs. Also there are servers include thousands of GPUs to be used together on the cloud.

By a quick prediction it can be said with a good supercomputer setup we can get 10 millions passwords test per second performance. Unfortunately, for all available passwords set, still 10 millions is not good enough to be practical.

It seems there are two ways to make password cracking solutions practical:

a. Narrowing dictionary up to some user behaviour or clues about password setter
b. Make a setup to let end users to permit your application do cracking distributed

Narrowing the dictionary is not easy because nobody can be sure that the real password is in the remaining. This might be an option for governors and intelligent agencies since they might collect some information about password setter. Even for governors this option might be only resource consuming without any result.

On the other hand, todays, there are too many different project which use end user resources and share profits from the job. In recent years, especially with expending of blockchain technology, more than ten projects appeared with this business model. Also, because of the cryptocurrency mining, people already have bought powerful GPUs and most of them not used effectively. Those GPUs might be used for password cracking which may result in worldwide cracking.

REFERENCES

Periodicals

Armin Ahmadzadeh, Omid Hajihassani, Saeid Gorgin, “A high-performance and energy-efficient exhaustive key search approach via GPU on DES-like cryptosystems”, published in Springer Science+Business Media, LLC, 2017

Other Publications

J. Ma, X. Chen, R. Xu, and J. Shi, “Implementation and evaluation of different parallel designs of aes using cuda,” in Data Science in Cyberspace (DSC), 2017 IEEE Second International Conference on. IEEE, 2017, pp. 606–614.

Michael Hofmann, Robert Kiesel, Dirk Leichsenring, Gudula Rünger, "A Hybrid CPU/GPU Implementation of Computationally Intensive Particle Simulations Using OpenCL", published in 17th International Symposium on Parallel and Distributed Computing (ISPDC), 2018

Yongseung Yu, Seokwon Kang, Yongjun Park, "Runtime Profiling of OpenCL Workloads Using LLVM-based Code Instrumentation", published in IEEE Region 10 Conference, 2018

Ayman Abbas, Rian VoB, Lars Wienbrandt, Manfred Schimmler, “An efficient implementation of PBKDF2 with RIPEMD-160 on multiple FPGAs”, 20th IEEE International Conference on Parallel and Distributed Systems, 2014

Andrea Visconti, Federico Gorla, “Exploiting an HMAC-SHA-1 optimization to speed up PBKDF2”, published in IEEE Transactions on Dependable and Secure Computing, 2018

Ning Li, Xiaojun Dang, Yang Zhang, "Realizing High-Speed PBKDF2 Based on FPGA", published in International Conference on Intelligent Transportation, Big Data and Smart City, 2015

Xiaochao Li, Chunhui Cao, Pengtao Li, Shuli Shen, Yihui Chen, Lin Li, "Energy-Efficient Hardware Implementation of LUKS PBKDF2 with AES on FPGA", published in IEEE International Conference Trustcom, 2016

Tesla V100 Nvidia, https://www.nvidia.com/en-us/data-center/tesla-v100/
OpenCL, https://developer.nvidia.com/opencl
SQLite File Format, https://www.sqlite.org/fileformat.html