Distributed Hash Cracker: A Cross-Platform GPU-Accelerated Password Recovery System

Andrew Zonenberg
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180
zonena@cs.rpi.edu

April 28, 2009

Abstract

Recovery of passwords protected by one-way hashes is a problem ideally suited to parallel computing, due to the embarrassingly parallel nature of a brute force attack. Although many computer forensics and penetration testing tools can perform multithreaded hash cracking on SMP systems, modern iterated-hash techniques require unacceptably long crack time on a single computer. The author is aware of only one system capable of brute-force hash cracking across multiple computers, an expensive commercial product which only runs on Windows and does not permit the user to extend it to support new algorithms.

This paper presents a distributed hash cracking system capable of running on all major platforms, using nVidia GPU acceleration where available. The cracker is modular and allows precompiled hash algorithms or "crack threads" (guess generation and test logic) to be added with no modification to the existing application binaries, in order to add support for new algorithms or make use of hardware acceleration. Linear scaling is demonstrated up to 64 processor cores. Performance testing was also conducted on larger clusters but due to their non-homogeneous nature it was not possible to achieve meaningful scaling results.

1 Introduction

In many situations (such as forensic analysis, data recovery, or penetration testing) it is necessary to recover the plaintext of a password encrypted with a cryptographic one-way hash: a function mapping arbitrary sized inputs to fixed sized outputs in such a way that the mapping cannot be easily reversed. One of the distingushing characteristics of a cryptographic hash, as opposed to a non-cryptographic hash function (e.g. CRC-32) is that it is designed to exhibit a strong avalanche effect (a single-bit change to the input will change, on average, a random half of the output bits).

Some hash algorithms, such as MD5, have been discovered to exhibit collision weaknesses [1] : it is possible to generate two messages which hash to the same value. This is generally an easier problem than the so-called "first preimage" attack, where an input value is calculated which hashes to a provided value. For the purposes of this project it was assumed that the target hash algorithm does not have a known first preimage attack and the only way to recover a plaintext password is a brute-force attack. [2] is an example of a typical program designed for recovering hashed passwords by brute force, which is capable of exploiting multicore parallelism in SMP systems but cannot run on more than one system at a time.

A password can be thought of as n symbols picked (with repetition allowed) from a c-symbol "character set". Simple mathematics shows that there are n^c possible passwords. On average, half - $\frac{n^c}{2}$ - will need to be tried. Increasing either n or c will raise the number of combinations exponentially, as shown below:

c	n	$\frac{n^c}{2}$
26	6	154,457,888
26	7	4,015,905,088
52	6	9,885,304,832
26	8	104,413,532,288
52	7	514,035,851,264
52	8	26,729,864,265,728
62	8	109,170,052,792,448

Precomputation attacks, such as rainbow tables [3] are feasible against some hashing algorithms. Many applications, such as Unix-like operating systems, use "salted" hashes: a randomly generated value, which need not be kept secret, is combined with the password during hashing to increase the amount of time and storage needed for table generation. A well-designed salting algorithm can leave a brute-force attack as the only viable means of recovering a password.

Even at a 40 million hashes per second - possible for MD5 on a moderately fast multicore system using [2] - performing a brute force attack on a decent password is extremely time consuming. For a typical password of 8 characters drawn from the set a-zA-Z0-9, we have $c = 62$ and $n = 8$. This would take an average of 2,729,251 seconds, or just over a month, to break on a single computer. Some systems, such as Linux/FreeBSD MD5crypt or GPG file encryption, perform multiple iterations of the hash function (typically > 1000) to slow down brute force attacks - potentially increasing the crack time for our hypothetical password to over 83 years!

Luckily, this problem can be easily parallelized by partitioning the search space between multiple systems. In theory, linear scaling can be realized due to the complete lack of dependencies between blocks of search space: 31 equivalent computers would be able to break our MD5 in only a day, and by adding more systems (or making use of GPU acceleration) crack time could potentially be reduced to several hours.

2 Related work

Several studies, most notably [4, 5], have examined the feasibility of parallel hash crackers, but nearly every system was a "classic HPC" application built using the Message Passing Interface (MPI). These systems typically used a static set of CPU-based compute nodes connected in a homogeneous cluster, lacking GPU acceleration or the ability to recover from serious errors (such as a single compute node crashing).

[6] is a TCP/IP based parallel hash cracker capable of GPU acceleration, however it is a closed source commercial product which cannot be studied at the source code level, does not support addition of new hash algorithms or salting algorithms, and only runs on the Windows operating system. The author was unable to locate any cross-platform parallel hash crackers capable of utilizing GPU acceleration.

3 System Architecture

3.1 Overview

Our distributed cracker uses a relatively standard master-slave design: a central master server, responsible for coordinating the overall crack effort, and one or more compute nodes, which perform the actual cracking. TCP sockets are used for communication between compute nodes and the master. As of this writing MD5 was the only hash algorithm fully implemented. MD5-based shadow hashes and SHA-1 are in progress, and LM/NTLM are planned for the near future.

The cracker is being developed as an open-source application (BSD licensed) by RPISEC, the computer security club at Rensselaer Polytechnic Institute. Interested parties may download source at http://rpisec.net.

3.2 Master server

The master server (written in C++) serves two functions: it provides the user interface from which a crack is actually launched, and schedules units of search space to each compute node for processing.

During initialization, the master server spawns a networking thread, which hosts a socket server for communicating with compute nodes, and then goes into an input loop, waiting for the user to type a command. Meanwhile, whenever a compute node connects, a separate thread is spawned to service it. At any time, the user may type an informational command (such as "stats", which prints out the number of connected compute nodes and, if a crack is in progress, the portion of the search space covered so far), a configuration command (such as "set charset aA", which selects the case-sensitive alphabetic character set), or a crack command (such as "crack md5 900150983cd24fb0d6963f7d28e17f72").

Once a "crack" command has been issued the master enters a loop, allocating a work unit to each compute node in a round-robin fashion until all work has been allocated, blocking if no nodes are available. The master keeps track of the work unit each compute node is allocated; if the TCP connection to a compute node is dropped its WU is returned to the pool and given to the next available node. New compute nodes may join a master server at any time; if a crack is in progress the new node will be given the next available work unit.

If all work units are completed with no success reports, the crack is declared to have failed (not in the specified search space). On the other hand, if a node reports success the master will display the cracked hash and return to the idle state. Work units in progress on other nodes are allowed to complete: to speed processing and simplify the system design a work unit cannot be aborted over the network once started.

3.3 Compute node

3.3.1 Overview

The compute node (written in C++) is responsible for performing the actual work of a crack. When started, it connects to the master server and announces its capabilities. It then searches the current directory to find DLL or SO files containing hash algorithms or crack threads, loading and initializing any that are found.

When a work unit is received, the compute node parses it and spawns a crack thread for each computing device (processor core or CUDA GPU) in the system. Each crack thread is assigned an equal fraction of the work unit in the current release; future versions will benchmark each device and determine the optimal division of labor. (The current version does not support mixed CPU and GPU cracking, precisely for this reason.)

3.3.2 CPU implementation

The CPU crack thread consists of a loop over the search space, generating a set (1 or 4, depending on the hash in use) of candidate values, hashing them, and then testing the results. In the current version, hashing is a separate function stored in a DLL/SO (to support pluggable hashes) and invoked from the generation and test code in the crack thread. We are considering merging the generation and test code into the a single monolithic "crack unit" to eliminate function call overhead, as was done with the CUDA version.

3.3.3 CUDA implementation

The CUDA crack thread divides the search space into blocks which are small enough to be processed in a few hundred milliseconds or less. The best thread count for this algorithm on this hardware is then looked up from a cache file (if the value is not found, a benchmark is conducted to calculate it) and a kernel is launched to process the block. The kernel performs guess generation, hashing, and testing in a single unit to reduce memory bandwidth and avoid the overhead of kernel switching, at the cost of additional code space due to repeated generation and test logic.

Earlier versions of the CUDA design used a pipelined design similar to the PS3 - a kernel which generated a block of guesses and saved them to GPU memory, followed by a kernel which hashed the block, followed by another which tested the results. When this design was discovered to be memory-bound, the design was switched to the current monolithic kernel. This resulted in a substantial performance increase.

3.3.4 PS3 implementation

The Cell crack thread was designed for the PS3, and thus uses only six of the eight SPEs on the processor. The current design consists of two parallel pipelines (each handled by a separate PPE thread) of three SPEs each.

The first step of the pipeline generates a set of candidate values, then sends them to the second via DMA transfer and begins producing the next block as soon as the DMA has finished. The second stage hashes the inputs and then DMAs them to the third for testing in the same manner. The test stage then compares the data against the target hash and reports success if found.

While this architecture works - and can crack 20 million MD5s per second on a PS3 - it appears to be getting memory bound. Borrowing an idea from the CUDA implementation, we plan to switch to a monolithic "crack block" containing repeated generation and test code for each hash, in order to permit all data of interest to be kept in registers rather than being moved to local storage.

3.4 Network protocol

In order to ease debugging and avoid endianness issues, it was decided to use a simple text-based protocol loosely modeled on HTTP. Strings are transmitted in a modified Pascal format: the length in bytes as ASCII decimal, followed by a space, then the string.

A work unit consists of a method (always "crack" in the current version), followed by the target and a newline character. Additional data (such as character set or guess ranges) is communicated with "headers" in HTTP format: the name of the header, a colon, a space, and the value. Example work unit:

```
CRACK 32 a920d3b22d35e528e4b52a244cc00328
Algorithm: 3 md5
Charset: 26 abcdefghijklmnopqrstuvwxyz
Start-Guess: 3 aaa
End-Guess: 3 zzz
```

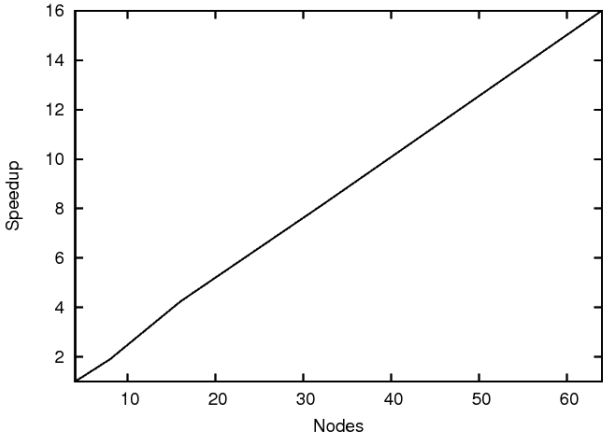
Once a work unit is completed, regardless of success, the compute node contacts the master to indicate the current situation. Valid responses are "continue" (search space covered, target not found, ready for next work unit), "leaving" (search space covered, target not found, compute node is terminating), and "found" (target hash was cracked, cracked value is on the next line of the response).

4 Performance Results

4.1 Scaling

A scaling test was conducted on a cluster of 2.0 GHz Opteron systems running Linux. The cracker scaled very well up to the limits of the test (64 processors), slowing down slightly at 8 nodes and then achieving barely superlinear speedups for the rest of the runs. These anomalies appear to be due to measurement error.

CPUs	Speed (x1M hash/sec)	Speedup
4	15.88	1
8	30.35	1.91
16	67.35	4.24
32	128.79	8.11
64	254.29	16.01



4.2 GPU vs CPU performance

No formal scalability testing has been conducted on GPUs as of this writing, because the author was unable to obtain a large number of identical cards for testing on. Early performance results are promising, however, and suggest that a few of GPUs will be able to outperform a moderately sized CPU-based cluster.

System	Speed (x1M hash/sec)
P4 2.0 GHz	3.24
Core 2 Duo 2.2 GHz	16.19
GeForce 8600M GT	24.42
2x quad Xeon 2.0 GHz	55.33
Quadro FX 4600	102.26
GeForce GTX 260*	250

* tested on earlier version of code

4.3 Throughput tests

The highest throughput reached as of this writing was 1.88 billion MD5s/sec during a ten-minute test on the following systems:

- 47 Dual Xeon 2.8Ghz (Dual Core)
- 14 Dual Xeon 2.8Ghz (Quad Core)
- 29 Pentium D 2.8 (Dual Core)
- 15 AMD X2 5000+

This is substantially less that the theoretical capabilities of the cracker on this hardware, however this test was conducted by a third party and the exact parameters of the test are unknown. It is believed that these systems were in use by other applications during this test, making its validity as a scaling measurement questionable. However, as a demonstration of what a distributed cracker can do given adequate hardware, it appears to have served its purpose. (An MD5 of a "standard good password" - 8 characters, case sensitive alphanumeric - could be cracked in an average of 16 hours on this system. Eight characters single-case alphanumeric would last a mere 12 minutes.)

5 Conclusions

The use of a distributed brute-force attack for recovering hashed passwords appears very feasible for any password of ≤ 8 characters length using common character sets (i.e. alphanumeric case sensitive). For only a few tens of thousands of dollars, one can build a cluster capable of breaking most common passwords (assuming a non-iterated hash) in hours. It is likely that a large enterprise (e.g. data recovery service) with a multi-million dollar budget could scale such a system up to several thousand multi-GPU nodes and break a typical password in minutes.

Iterated hashes, such as the md5-based crypt() used in current Linux and BSD operating systems, will substantially slow down attacks, but do so by a linear factor. Although we have not yet performed large-scale testing of MD5crypt (due to the lack of CUDA or optimized x86 implementations), we have no reason to expect its performance to differ from that of unsalted MD5 by more than a linear factor.

6 Future work

CUDA is not the only general-purpose GPU computing platform around. Due to time limitations it was not possible to explore ATI Stream Processing and similar platforms.

In order to be useful for penetration testing or commercial password recovery, the cracker will need to support most popular hash algorithms. The current version does not support some (i.e. NTLM) at all, and others are only partially implemented (MD5crypt does not have CUDA, Cell, or x86 assembly implementations). We plan to continue working on these in the near future.

The current system maintains a socket and thread for each connected compute node. At node counts in the thousands or higher, file handle limitations will begin to cause problems. We are currently exploring stateless protocols which should permit much better scaling to extreme node counts.

Our Cell code has significant room for improvement, as this was a relatively recent addition to the project. Rates of up to 80 million MD5 hashes per second have been cited on a PlayStation 3, while our code only reaches a quarter of that.

7 Acknowledgements

The author would like to thank Dr. Chris Carothers, Rob Escriva, Ryan Govostes, Alex Radocea, and the members of RPISEC for technical advice, code contributions, and computer time. Compatibility and performance testing would not have been possible without donations of processing time from Ryan MacDonald, Louis Peryea, Andrew Tamoney, Jeff van Vranken, and Chris Wendling.

References

[1] Sotirov et al, "MD5 considered harmful today: Creating a rogue CA certificate" [Online document] [Cited 2009 Apr 20], Available HTTP: http://www.win.tue.nl/hashclash/rogue-ca/downloads/md5-collisions-1.0.pdf

[2] "MDCrack, bruteforce your MD2 / MD4 / MD5 / HMAC / NTLM1 / IOS / PIX / FreeBSD Hashes" [Online document] [Cited 2009 Apr 9], Available HTTP: http://membres.lycos.fr/mdcrack/

[3] Oechslin, P. "Making a Faster Cryptanalytic Time-Memory Trade-Off", 2003

[4] Bengtsson, J. "Parallel Password Cracker: A Feasibility Study of Using Linux Clustering Technique in Computer Forensics", Digital Forensics and Incident Analysis, 2007.

[5] Lim, R. "Parallelization of John the Ripper (JtR) using MPI" [Online document] [Cited 2009 Apr 9], Available HTTP: http://www.ryanlim.com/personal/jtr-mpi/report.pdf

[6] "Elcomsoft Distributed Password Recovery" [Online document] [Cited 2009 Apr 9], Available HTTP: http://www.elcomsoft.com/edpr.html