2008

High performance password cracking by implementing rainbow tables on nVidia graphics cards (IseCrack)

Russell Edward Graves
Iowa State University

High performance password cracking by implementing rainbow tables on nVidia graphics cards (IseCrack)

by

Russell Edward Graves

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-majors: Information Assurance, Computer Engineering

Program of Study Committee:
Doug Jacobson, Major Professor
Thomas Daniels
Cliff Bergman

Iowa State University

Ames, Iowa

2008

Table of Contents

List of Figures

Abstract

IseCrack is a high performance implementation of rainbow tables on nVidia graphics cards (GPUs). It explores the limits of current technology in password cracking, and demonstrates the vulnerability of non-salted passwords to high speed GPU-accelerated attacks, using commercial off the shelf hardware.

Passwords are by far the most common authentication method for users, and many users utilize the same password in multiple places. Many systems, including all current Microsoft operating systems, utilize non-salted passwords. If these passwords are vulnerable to attack, a user's encrypted files and online accounts can be accessed.

IseCrack demonstrates that very high speed attacks against non-salted hashes are feasible, and highlights the necessity for salted password stores. IseCrack achieves a 100x speedup over existing implementations on inexpensive easily available hardware, and is designed to scale to large clusters.

Chapter 1: Overview of IseCrack

In today's world, passwords are used for authentication of users in almost all applications, including local machine accounts, domain accounts, and web. These passwords are stored with varying degrees of security, including plaintext (stored as the password), simple hashes of the password (LanMan, NTLM, MD5), and salted passwords with multiple iterations (Unix implementations, good web applications). While the hash provides a layer of security for the password if the hashes are compromised, modern high performance implementations of password cracking systems can defeat non-salted hashes quickly through pre-computation attacks (using large amounts of previously calculated data to rapidly crack the hash), and can attack salted implementations that were previously considered to be secure due to the amount of computation required to successfully attack them.

General Purpose Graphics Processing Unit (GP-GPU) computation has also arrived on the computation scene within the past two years[11], allowing execution of code on the massively parallel stream processing hardware present in modern video cards, with performance previously considered to be firmly in supercomputer territory. A top of the line video card is capable of roughly 1 TFLOP under ideal conditions[1], as compared to 20-30 GFLOPs for a modern Core 2 Duo[2]. Modern GPUs are able to rapidly process integer operations as well as floating point operations, and are well suited to massively parallel problems.

Password cracking and rainbow table implementations are problems that fit very well within the massively parallel problem domain, and are candidates for acceleration using GPUs. As there are no existing public rainbow table implementations running on GPUs, and

the existing CPU-based rainbow table implementations use very slow reduction functions that do not function efficiently on a GPU (or, arguably, on a CPU), the entire rainbow table cracking system is re-implemented as a GPU-accelerated project to determine what performance is achievable, and what this means for user password security.

Chapter 2: Overview of Rainbow Tables

Rainbow tables are a pre-computation based approach to reversing hashes. They require a large amount of pre-computation, but can store the results of this in a reasonable amount of space. When searching for a hash, additional computation is required, but the computation required for searching is significantly less than the amount required for the pre-computation, and significantly less than the amount required to brute force a password.

By generating long chains of passwords and hashes, tied together by the hash function and a reduction function (described in detail in the implementation section), rainbow tables store a compressed representation of a password search space. By performing similar computations on a provided hash, they are able to dramatically reduce the amount of computation required to find the original password. As with many algorithms, there are limitations with rainbow tables. Unlike a brute force algorithm, they are not guaranteed to find a password within the search space, as the algorithm is probabilistic in the coverage of the password space, and a password will only be found if it is represented in the generated tables. However, very high success probabilities can be achieved, and the search time is significantly less than with a brute force algorithm. The details of rainbow table operation and the IseCrack implementation are covered in a later section.

The crack time/storage space tradeoff of rainbow tables is adjusted by changing the chain length. Longer chains require less storage space, but require more computation (and more time) to crack passwords.

Chapter 3: Overview of password systems and attacks

One way hash functions

The vast majority of modern authentication systems use one-way hash functions to store passwords. This allows a representation of the password to be stored without storing the actual plaintext password. If an attacker compromises the password store, they are unable to view plaintext passwords, and instead see the hash (as well as any other information stored with the hash, such as a salt value).

A one-way hash function is a function that takes an input of any length, and converts it into a fixed length output string. The properties of this function are such that there is no currently known direct way to reverse a given hash output into one or all of its possible inputs. As hash functions are often used in cryptography and for verifying file integrity during and after transfers, most hash functions have the additional property of being very fast. The one-way properties of the function are good for secure password storage, but the high speed is not ideal for secure password storage, as an attacker can iterate through the password space at the same high speed.

Salted password storage

There are two primary methods of storing a hashed password: salted, and unsalted. Unsalted passwords are those produced when the password storage system takes the provided password, runs it through a defined hash function, and stores the output. This is easy to implement, and is used on many websites, as well as all current versions of the Windows operating system. The significant flaw with unsalted password storage is that a hash value will always correspond to a specific password (figure 1).

If a password results in a certain hash on one system, this same password to hash relationship will be true for all other users on the system with the same password, and for all

```
          Unsalted password scheme
System 1, user a, password abc: 0x1234
System 1, user z, password abc: 0x1234
System 2, user m, password abc: 0x1234
```

Figure 1: Unsalted passwords

users on other systems with the same password (and the same hash routine for password storage). This property allows a pre-computation attack to be done, in which passwords corresponding to certain hash values are calculated ahead of time. If an attacker gains access to the hash store, they are able to use pre-computed data to speed the computation involved in reversing the hashes back into the source passwords.

Salted password involve the use of additional random data when hashing the password (figure 2). This random data is appended to the password before it is run through

```
                        Salted password scheme
System 1, user a, password abc, salt 45hx: 0xC142
System 1, user z, password abc, salt a5kz: 0x14FA
System 2, user m, password abc, salt 1234: 0xC1D3
```

Figure 2: Salted passwords

the hash algorithm and is then stored with the password. As a result, if multiple users on the same or different systems have the same password, they will have different hashes. The other beneficial component is that the salt effectively expands the password space. While attacking an 8 character password is relatively easy, a 32-byte salt requires an attacker without knowledge of the salt to attack a 40-character password (well beyond current computational abilities).

While attackers can still gain the password hash/salt information and attack the system if they compromise the hash store, a pre-computation attack is no longer effective. The attacker must attack each password separately with knowledge of the salt, a much slower process.

Time/space tradeoffs in password cracking

Password cracking requires resources. In general, cracking passwords requires both time and space to attack hashes and determine the password that generated them. Two approaches (brute force attacks and full pre-computation) exist at the ends of the time/space tradeoff spectrum, and rainbow tables exist in the middle, with their exact position being dependent on the parameters used to generate the tables.

The standard password attack, brute forcing, requires no storage space (beyond wordlists if they are used), but requires significant amounts of time. A brute force attack iterates through the password space, hashing each password, and looking for a match with the provided hash or hashes. Brute forcing a password does not require any pre-computation, does not require significant storage space, and does not save the results of the computation performed beyond the passwords found (if any). A brute force attack is the only feasible attack against a salted password system, but it rapidly runs into limitations of the size of password space that can be brute forced in a reasonable period of time.

The other end of the spectrum is full pre-computation. With this approach, all possible password and hash combinations are calculated ahead of time and stored in tables, sorted by hash order. This allows very rapid searching, but requires extremely large amounts of storage for even a modest password space (figure 3).

```
            Password space sizes, 16 byte hashes
8 characters, 95 element charset: (95^8) * 24 ~= 141 PB
6 characters, 95 element charset: (95^6) * 22 ~= 15 TB
10 characters, 26 element charset: (26^10) * 26 ~= 3 PB
```

Figure 3: Password space sizes

Rainbow tables exist in the middle, between brute forcing and full pre-computation. A large amount of pre-computation is done to speed searching through the password space, but the results are stored in a highly compressed form. By altering the chain length both the storage space requirements and the search speed can be modified. A longer chain length will provide better compression, but will also take longer to search. The total pre-computation time is not affected by the chain length.

The same 8 character password set that takes 141PB to store in it's entirety, stored in chains of length 1,000,000, only requires 151GB to represent – a much more reasonable amount of storage. Storing in chains of length 100,000 will require 1.5TB, again a feasible amount of storage. However, the table with length 1,000,000 will take significantly longer to search than the table of chain length 100,000 due to the candidate hash generation stage. As the speed of hardware increases, longer chain lengths can be used to represent larger password spaces, while staying within the limits of the available storage technologies. Additionally, as faster hardware becomes available, the previously generated tables can be searched more rapidly, while at the same time creating new tables that fully utilize the performance available in the new computation hardware.

Chapter 4: Video Cards as General Purpose Processors

In recent years, video cards have progressed beyond fixed function video display devices to allow a wide variety of code to be executed on them. The pixel shader pipeline allows a series of operations to be performed on each pixel in the output. Originally, these operations allowed for new effects, such as bump mapping, and per-pixel lighting, shading, and coloring[9]. As the pixel shaders advanced, they slowly evolved into fully programmable pipelines, able to run arbitrary code, and video card makers realized that they were very few steps away from a parallel stream processor. The new generations of both nVidia and ATI cards support this use, and both brands of cards are now usable as full stream coprocessors for certain workload types.

nVidia and ATI have both released APIs within the past two years that allow general purpose code to be run on the GPUs. While both are usable, nVidia's Compute Unified Device Architecture (CUDA) API is significantly more advanced, and is being actively developed and supported. nVidia cards were also already present for use in several systems. For these reasons, nVidia cards were chosen for the initial implementation. Extending the code to ATI graphics cards may be explored in the future.

SIMD (Single Instruction Multiple Data) and SIMT (Single Instruction Multiple Thread)

SIMD and SIMT refer to programming and execution models that have a single instruction stream operating on multiple elements of data at once. While many SIMD implementations exist (SSE, Altivec, many game consoles), CUDA uses a variation known as SIMT. This involves a single instruction stream running multiple threads. The threads have their own local data and are scheduled independently of one another (although the

threads are executed with other threads in their block). This allows a programmer to create a large number of threads (10,000 or more threads is not uncommon, and 100,000 threads is possible), and allows the hardware and thread scheduler to deal with the scheduling. The advantage of this approach is that with a large number of threads ready to run, the scheduler can switch between ready threads to hide memory latencies, and can schedule threads to run on as many processors as the card has available. The programmer, in general, can write the same code for low powered laptop video cards as well as high end desktop gaming or workstation powerhouses. The primary disadvantage is that the programmer has no guarantee of any execution order, so the thread execution must be able to proceed in any order. There are synchronization and communication primitives that can be used, but they can affect execution speed.

nVidia CUDA View of hardware

From the perspective of the CUDA programming API, a modern video card contains many blocks of stream processors. Each block consists of a number of stream processor units. The stream processor units are what actually execute the code. They have their own registers and access to per-block shared memory. The primary limitation is that there is a single instruction dispatch unit for each bank of stream processors. If all the threads are running the same code, the stream processors are able to execute in parallel, and are quite fast. However, if the code branches, execution has to serialize, processing each section in turn, which dramatically slows processing throughput. Additionally, the stream processors are very simple processors, lacking lookahead, speculative execution, branch prediction, or any other "modern" features. This allows the bulk of the transistors to be spent on execution

hardware as opposed to support hardware. A resource dependency is resolved by stalling the pipeline until the dependency has been resolved.

While many problems can be accelerated using GPUs, GPUs excel at completely parallel problems: performing the exact same operations on different blocks of data, with no inter-thread communication. Rainbow tables and password cracking are perfect examples of this class of problem, and, as such, can be sped up dramatically through proper implementation on a GPU.

GPU-specific considerations

This project is being implemented on GPUs with a goal of extracting maximum performance from the hardware. As such, understanding the strengths and weaknesses of the GPUs is vital for extracting maximum performance. Several of the important considerations for obtaining maximum performance are:

- GPU performance drops dramatically if the code branches. While in some cases branches are unavoidable, keeping them to a minimum is critical for extracting maximum performance. IseCrack's code minimizes the use of branches. The only place branches are used is for the main loops, and to test if generated hashes are equal to the provided hash. To help reduce the impact of branches, the code is structured such that branches only occur if absolutely needed. When comparing hashes, if none of the initial words of the hashes match, none of the subsequent words are checked, and the code does not branch at all.
- CPU and GPU code can overlap execution. After the GPU kernel is launched, the CPU can continue to execute code. It will execute independently until the next GPU

call, at which point the CPU will wait for the GPU to finish. If the GPU finishes before the CPU is done, the GPU will idle until the CPU is done. This allows for "free" preprocessing and postprocessing of GPU data if it is done properly and if the CPU section completes before the GPU code. This is used in the table generation code (which makes up the bulk of the execution time for the project) in order to keep the GPU running constantly. Effectively, the sorting and network transmission can be done “for free,” as the CPU is not being used to generate hashes.

- GPUs do not make function calls efficiently, and cannot recurse. To keep things as efficient as possible, the compute kernels are written without function calls as a single code block.
- Certain functions are very fast on GPUs, and other functions are not. Integer division and modulus are particularly slow, due to fewer execution units assigned to these functions and the complexity of the operations. 64-bit integer division and modulus are particularly slow and should be avoided if at all possible.
- GPUs have a limited number of registers available to use. In order to fully utilize the stream processing units in a block, a thread can use no more than 10 registers. The more registers that are used, the fewer threads that can be run in parallel (physically – virtually, they are all being run in parallel). Keeping register counts as low as possible is vital for performance. However, register usage can be used to prevent memory access, and so code may be overall faster with an increased register count but significantly reduced memory access. Balancing these tradeoffs is part of the performance tuning process.

- Global memory accesses are very slow (400-600 cycle latency). These latencies can be hidden if there is sufficient compute code ready to run in other threads, or if the load is performed sufficiently in advance of the data use. Writes to global memory do not block, but with the undefined thread execution ordering, using global memory for communication requires synchronizing and is best avoided.
- Local shared memory is very fast (in the best case, as fast as registers), but small (16k per thread block) and requires specific memory access patterns to fully utilize the available bandwidth without serializing accesses and slowing execution down. The shared memory is arranged in 16 banks, each 1 word (32 bits) wide. All banks can be accessed in parallel, but only one element per bank can be accessed in a given cycle. Data that will be accessed regularly should be copied to shared memory for access if it is not being stored in registers. Local memory also supports broadcast reads, where multiple threads reading the same address will receive the data in parallel.

Chapter 5: Previous Work & High Performance Password Cracking Systems

Work has been done previously with both rainbow tables and with GPU accelerated password cracking. However, the two have not previously been combined in a publicly available product. There are several commercial and free products available that implement parts of this project. A brief overview of the products and their strengths and weaknesses follows.

Rainbow Crack[3]: Rainbow Crack is considered to be the reference implementation of rainbow tables. It is a set of programs that allows for the creation, sorting, and searching of rainbow tables. It also uses a very "correct" reduction function, generating high quality password distributions in the chains. The primary flaw is that the reduction function is extremely slow, consisting of a large number of integer divides and modulus operations. It is implemented for general purpose CPUs, and while there have been attempts to accelerate the table generation with GPUs [12], the limitations of the reduction function prevent a fast GPU implementation. Additionally, to allow for the greatest effectiveness in GPU acceleration, the other functions must be accelerated as well.

ElcomSoft Brute Forcers[4]: Elcomsoft is a Russian software company making a variety of GPU accelerated brute forcers. They also have a $5000 product allowing for distributed password cracking. However, they do not currently use rainbow tables.

Free Rainbow Tables[5]: The Free Rainbow Tables project is a moderate sized distributed project that creates a variety of rainbow tables. They offer an online cracking service, as well as a download of the tables for offline personal use. While the tables generated are large,

they are still using the Rainbow Crack algorithms, and performance is due to the large number of CPUs processing as opposed to an efficient algorithm. They are effective at smaller password sizes, but do not appear to be currently scaling to GPU acceleration.

OphCrack[6]: OphCrack is a fast rainbow table based password brute forcer. However, like the others, it is CPU-only. The large pre-computed tables are available for sale, but these tables are still limited compared to what can easily be computed with a GPU accelerated system.

BarsWF[7]: BarsWF is a set of high performance GPU accelerated password crackers. They are quite fast, utilizing both heavy SSE4 optimizations for the CPU implementation as well as utilizing as many GPUs as are present in the system. The primary restriction is that they only search for one hash at a time, and like all other brute forcers, do not make any use of the calculations after the password is found.

After researching the available software, it was clear that no one was going forward with implementing a fully GPU-driven rainbow table implementation. ElcomSoft appears to be the most likely to release a rainbow table implementation for GPUs, but they are not transparent about their future intentions. The Free Rainbow Table project is another potential source of GPU accelerated rainbow tables, but discussions with the site administrators indicates they do not have anyone with the needed skills to implement the solution, and are looking at other options for performance gains on modern CPUs. The primary developer of BarsWF intends to create a distributed cracking system[8], but has no current plans to implement rainbow tables.

Based on these findings, it was clear that the way to test performance of GPU-accelerated rainbow tables would be to write an entire implementation from scrach, optimized for GPUs, and test performance with it.

Chapter 6: IseCrack Rainbow Table Implementation

Rainbow tables are a pre-computation based attack on hashes. They are a time/space tradeoff, compressing the data size of the pre-computed tables to a feasible amount of storage. They do require very significant amounts of up-front time to compute the tables, but after the tables have been created, they allow rapid cracking of any password represented within the tables. Unfortunately, rainbow tables are probabilistic in nature, so a crack can not be guaranteed, only determined to a certain level of likelihood (often in the 99.9% or higher range). To improve the cracking probability, multiple tables with different indexes are used. This allows for a higher crack probability within a given space. Also, perfected tables increase the crack probability within a given storage space, but require significantly more up front computation time.

To implement a rainbow table based attack on a certain hash, in addition to the hash function, a reduction function is needed. The reduction function takes a hash and turns it back into a password of specified characteristics (length, character set). The reduction function is critical to the operation of the tables, as a reduction function that generates bad results will prevent a usable table from being generated.

There are a number of different operational sections that go into the rainbow table generation and search process. The functionality of each section, and operational details, are described in the following sections.

Hash Function

NTLM hashes are a MD4 hash of the UTF-16 representation of the password. IseCrack is currently restricted to supporting ASCII characters in the crack character set. While NTLM hashes support the full Unicode character set, the vast majority of users restrict themselves to easily typed passwords on their native keyboard and are not willing to enter Unicode characters for each login. Additionally, the entire Unicode character space is far too large to attempt with current technology.

Password (ASCII): aBc (0x614263), 3 bytes
Unicode (UTF-16): 0x006100420063, 6 bytes
Little endian Unicode (UTF-16LE): 0x610042006300, 6 bytes

Figure 4: Password encodings (ASCII, UTF-16, UTF-16LE)

To prepare a password for hashing, it is converted from an ASCII representation to a Unicode (UTF-16) representation (figure 4). Additionally, it must be stored in little endian format (UTF-16LE), as this is the encoding used on x86 machines that run Windows. To do this, bytes of 0x00 are inserted after each ASCII character in the string to be hashed. This corresponds to the memory representation on a Windows machine. The length of the string to be hashed is twice the password length.

Once the password is represented in little endian Unicode, it is run through the MD4 hash algorithm to obtain the NTLM hash. MD4 is defined for any input length (including non-byte length inputs). However, for the purposes of rainbow table implementation, the MD4 function will only receive byte-length inputs, and further will only receive inputs up to a certain length. By removing unneeded code from the MD4 implementation, significant

speedups can be seen as long as the input is limited with certain constraints. Modifications and limitations of the MD4 implementation used for IseCrack include:

- Only one cycle is run. MD4 is defined for any length input, and each cycle processes 448 bits (56 bytes). However, for IseCrack, only a single cycle is implemented. This limits the input string length to a maximum of 55 bytes, or a 27 character password. This is well beyond what is currently feasible to calculate, and is not a limitation for the current project.
- Each password length has its own kernel with the length hard coded. This allows for reduced register count and elimination of length-related branches, both of which improve performance significantly by keeping the code path as parallel as possible.
- Input bytes that are always going to be zero (input words beyond the end of the password length) are hard coded to zero, as they will never have data. This allows for reduced register count and reduced register bank conflicts, which allow better performance. In certain functions, the compiler is able to detect these static conditions and automatically optimize without hand-optimization.

The end result is that, for each password length, there is an optimized MD4 function that generates correct output, for that specific input length, as quickly as possible. The reduction in register count allows for somewhat increased parallelization on shorter password lengths, and increased performance.

Reduction Function

The most important component of the rainbow table process is the reduction function. This function turns a hash back into a password. If this function does not generate an even distribution of passwords when used in chain generation, the entire rainbow table will be worthless. While a reduction function may perform well in a test environment with uniform input values, this does not guarantee proper performance in an actual rainbow chain. Due to the complicated interactions between the hash function and the reduction function, a function that generates good test data may, in an actual chain, generate highly "clustered" passwords, where certain passwords are very frequently represented and others are not represented at all.

Also of extreme importance, the reduction function must be fast on the hardware. Hash functions are designed to be fast, while a good reduction function is often slow. The Rainbow Crack reduction function, used in almost all current rainbow table implementations, involves significant amounts of 64-bit integer division and 64-bit integer modulus, and is quite slow on general purpose CPUs. It also does not translate well to a GPU-accelerated version, as 64-bit integer operations are some of the slowest operations on the card.

The reduction function used by IseCrack (figure 5) has been developed in several iterations. A password clustering problem, where certain passwords would be represented many thousand times and others would not be represented, was discovered late in development, when it was realized that many passwords that should be represented in the test tables were not. After generating test code to observe the password distribution, the clustering was observed. A new algorithm was written to generate more evenly distributed passwords. As an additional feature, the new algorithm supports arbitrary character sets,

which is useful for generating tables of longer password lengths but a smaller character set (characters 0-9 to password length 12, only lowercase or only uppercase to password length 10).

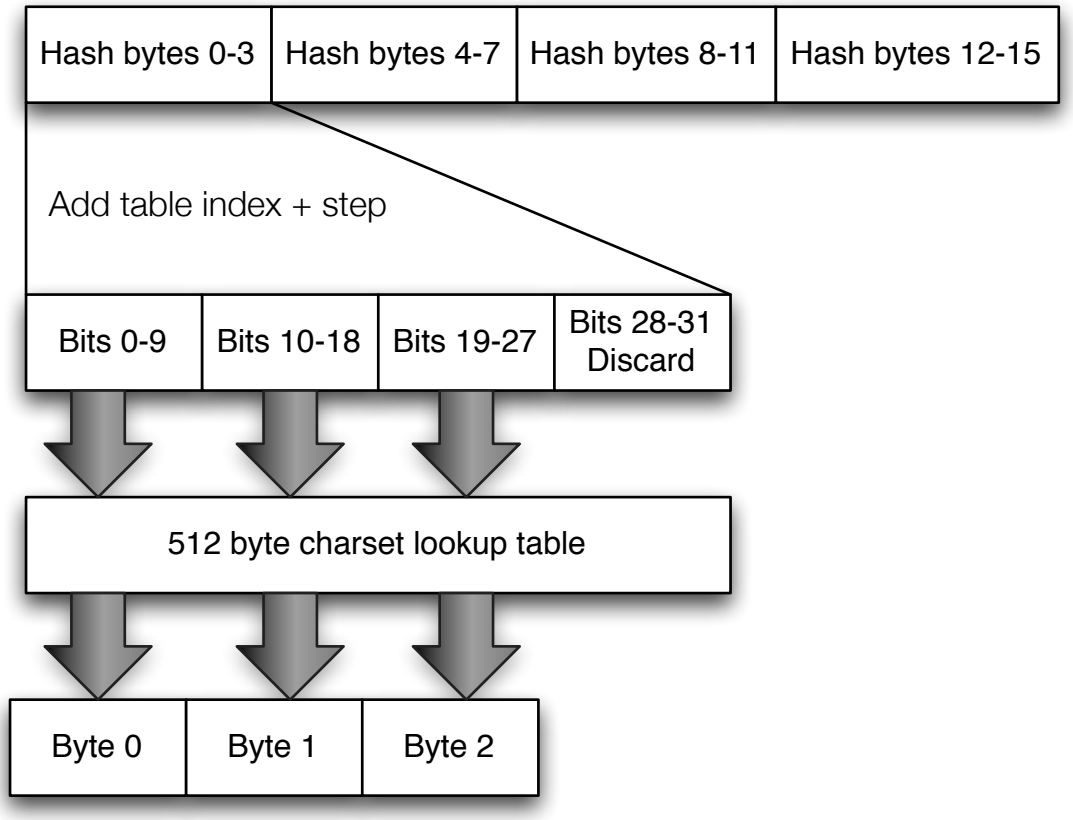

Figure 5: IseCrack reduction function

Each input byte from the hash has the table index and step added, to help prevent merging chains (described in the table generation section). After this, the lower 27 bits are stripped off and broken into three 9-bit segments. These are then used as indexes into a 512 byte character set table. A 1024 element table was implemented, but did not provide significant improvements in password distribution and was several percent slower. This reduction function relies on the speed of bit-masking and bit-shifting for power-of-two

modulus and division, and runs extremely quickly on the GPUs. The provided code (figure 6) is for length 8 passwords. Shorter passwords use fewer steps. As each register (b0, b1, b2, b3) is a full word (32 bits/4 bytes), two password characters are packed into each register. The higher position character is shifted by two bytes as it is inserted to place the characters appropriately.

```
z = (UINT4)(a+i+tableindex) % (512*512*512);

b0 = (UINT4)charset[(z % 512)];
z /= 512;
b0 |= (UINT4)charset[(z % 512)] << 16;
z /= 512;
b1 = (UINT4)charset[(z % 512)];

z = (UINT4)(b+i+tableindex) % (512*512*512);
b1 |= (UINT4)charset[(z % 512)] << 16;
z /= 512;
b2 = (UINT4)charset[(z % 512)];
z /= 512;
b2 |= (UINT4)charset[(z % 512)] << 16;

z = (UINT4)(c+i+tableindex) % (512*512*512);
b3 = (UINT4)charset[(z % 512)];
z /= 512;
b3 |= (UINT4)charset[(z % 512)] << 16;
```

Figure 6: IseCrack reduction code

One limitation of the current reduction function is that it wraps the character set as many times as possible into a 512 element lookup table. If the character set length is not a power of two, there will be an uneven distribution of characters. However, the speed of the current reduction function with this restriction is enough faster to make this tradeoff beneficial to overall system performance. Additionally, this reduction function will only work out to 12 characters (4 input words at 3 characters per word). However, as long as the same reduction

function is used in every step of the process, a different reduction function for longer passwords does not pose a significant problem to the operation of the system.

Table Generation

The table generation phase is the most time consuming phase of the rainbow table process. This phase can require GPU-years of computation time, depending on the password space being explored. To generate a usable set of tables, roughly 4 passes through the password space must be completed. For generating a set of optimal, perfect tables, closer to 40 passes through password space is required, but the tradeoff is significantly faster searching. The generate parameters can be adjusted based on the number of GPUs available and the desired performance.

The table generation stage involves the creation of the rainbow tables (figure 7). A rainbow table is simply a sorted collection of rainbow chains. Each individual element of a rainbow table (shown in the "Table Entry" boxes) is a rainbow chain, and represents a series of passwords equal to the chain length. To generate a rainbow chain, a random initial password is generated. This password is then hashed through the desired hash function. The hash is run through the reduction function to generate another password, which is hashed again. This repeats for each step of the chain (in this implementation, 100,000-1,000,000 times). After the computation, the initial password and end hash are stored. The initial password and end hash represent the entire computed contents of the chain, and this information is used later in the process to reduce the amount of computation required to find a password. The chain generation is done a large number of times to generate the rainbow tables.

One consideration is that, depending on the character set size, there may be many chains that merge - different input values leading to the same end value (chains 1 & 2 in figure 7). Chains may merge at any point in the chain. However, if chains have merged, there is a reduction of password space represented by them. The worst case, two chains merging after the first hash, means there are two functionally duplicate chains. If there are large numbers of merging chains, this wastes disk space, and impacts searching time. "Perfecting" tables involves removing all but one of the merging chains. This requires significantly more table generate time to cover a given password set, but produces dramatically more space and time efficient tables. This can be done on a per-table basis, as the subsequent functions work equally well on perfected or non-perfected tables. In figure 7, only chain 1 or chain 2 would appear in a perfected table.

The important innovation in rainbow tables is the use of a different reduction function for each stage of the chain. This is often implemented by passing the chain step into the reduction function. Note that in figure 7, ‘myPass’ appears twice, but as the hash is at different steps, it is turned into different passwords. Chains only merge if the same hash appears at the same step.

Tables also have an index value associated with them. This is used as another input to the reduction function that alters the password generation. Due to the probabilistic nature of table generation, an increasingly large table is more likely to have duplicate values. Getting very high crack probabilities with a single table requires much more disk space than using multiple smaller tables. The index value, as described above in the reduction function, is used on a per-table basis to alter the reduction function and improve overall efficiency.

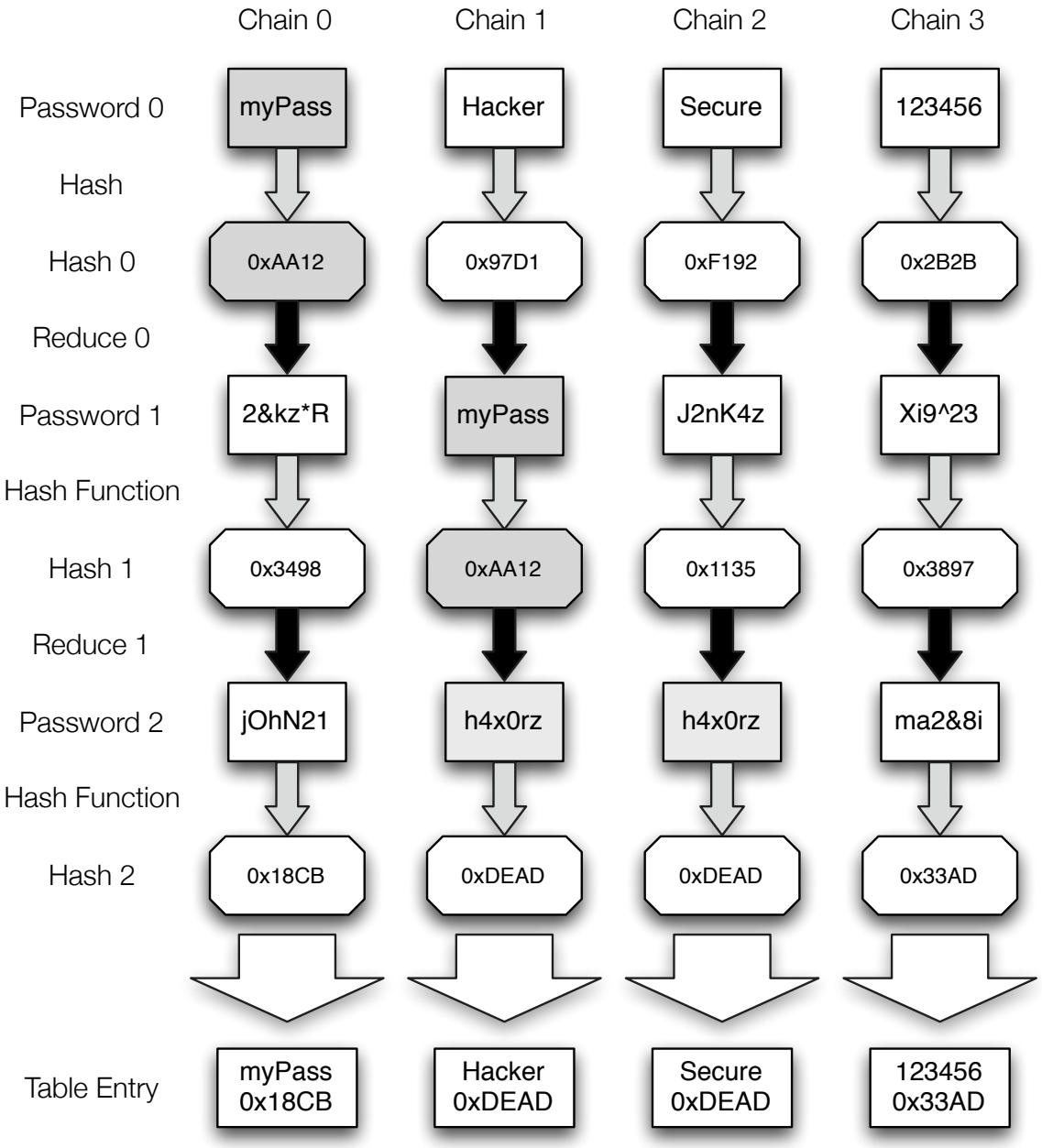

Figure 7: Rainbow table generation

IseCrack handles table generation by distributing table parts to remote compute clients. Data transmitted and received is described in figure 8.

```
               Table generation data packets
[Hash type]\n
[Character set]\n
[Detailed character set info or blank]\n
[Chain length]\n
[Password length]\n
[Number of chains to generate]\n
[Table index]\n

Client returns, for each chain, in sorted order:
[16 bytes: hash][16 bytes: password, null padded]
```

Figure 8: Distributed table generation packet

Generating the table parts involves several operations:

- Generate a random array of passwords. This is done on the host CPU using standard random functions while the previous set of hashes is being computed on the GPU.
- Computing the end hashes for each element in the initial array of passwords. This is the kernel run on the GPU.
- Sorting the end hashes for easier merging. This is done on the CPU, while the next set of hashes is being computed on the GPU. Quicksort is the algorithm used, as this is an efficient algorithm and executes quickly without needing significant additional memory space.
- Transmitting the sorted hashes/passwords over the network. This is done by the CPU while the GPU is processing.

On a fast network connection, the network overhead is not significant. However, if compute nodes are on a slower connection (residential DSL or cable), the network transfer

time may take as much time or longer than the GPU computation. Overlapping the network transfer with the GPU computation allows significant performance improvements. Also, while the sorting is a fairly fast process, it does take 5-10 seconds of CPU time for 1M elements. In figure 9, tasks that execute on the CPU while the GPU is processing are shaded. However, if the network connection is slow enough to not allow the complete transfer of data before the GPU kernel finishes, the GPU will sit idle until the transfer is done. As IseCrack is intended to run on a local high speed network, this is not a significant concern.

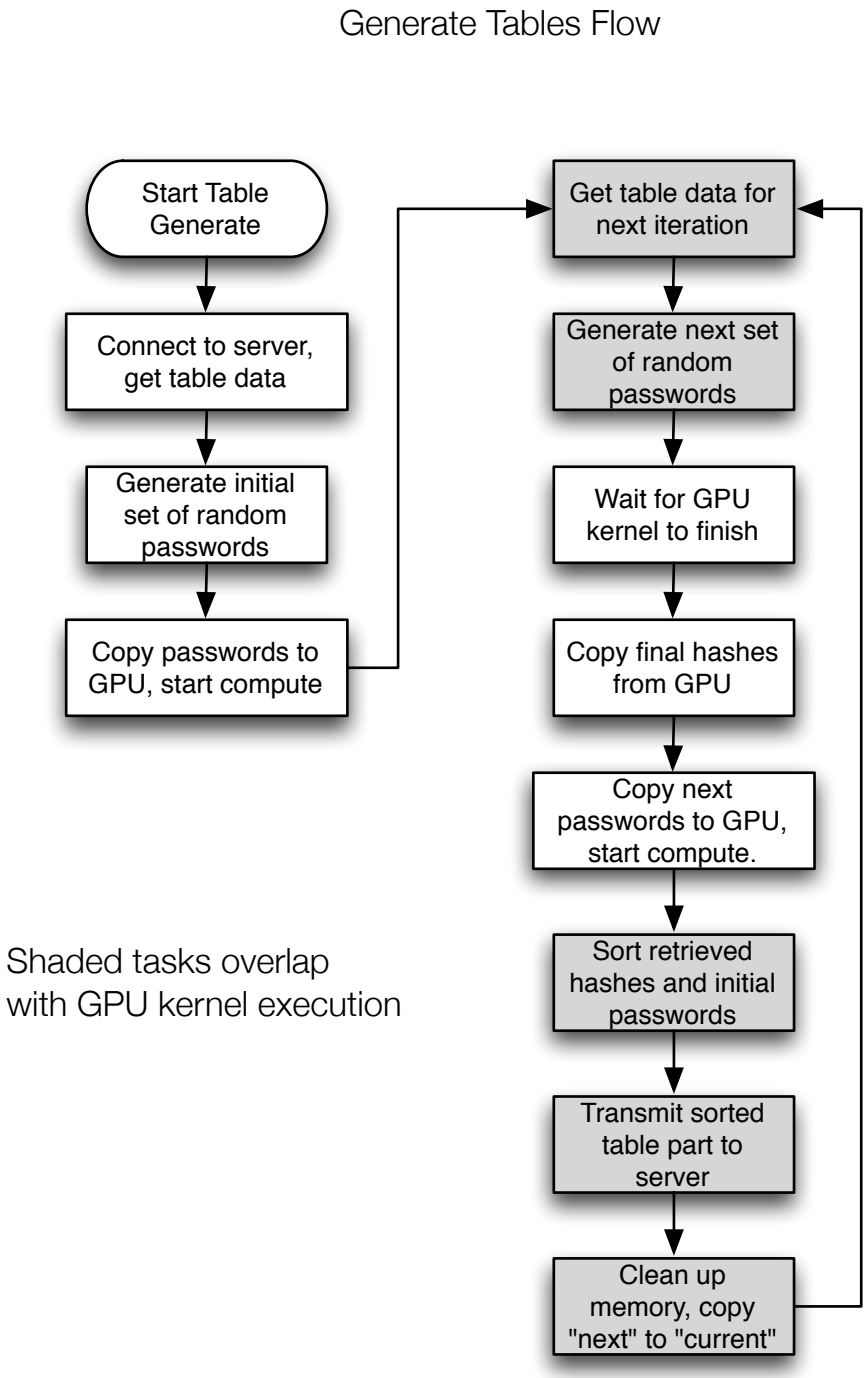

Figure 9: Rainbow table generation CPU/GPU overlap

Merging table parts and perfecting tables

Once table parts are created, they must be merged into a single large table. This process is done by reading all of the (sorted) table part files, and merging them together into a large sorted table. If the table is being perfected, the duplicate hash values are removed at this point, leaving only a single instance of each end hash in the generated table. The end result of the merge process is a large rainbow table, sorted by hash order.

IseCrack loads all the table part files as memory mapped files. This allows the kernel to handle the memory management of files. Additionally, when generating a perfected and a non-perfected table at the same time, the kernel will allow the open memory mapped files to be shared between two tasks and dramatically reduce disk read and memory requirements. Due to the size of the data files, this code must be run on a 64-bit machine and operating system, as the virtual memory space can exceed 4GB by very large amounts.

Once the files are loaded into memory, they are merged using a standard merge algorithm that picks the lowest value from the list of current positions. One drawback to this is that the compute complexity is O(number of elements * number of input files). Further optimizations could be made by first merging smaller numbers of chains into longer sorted chains, and then merging those together, as opposed to doing a flat merge of all input files.

If the table is to be perfected, this is done while merging the files. For a perfect table, the previously merged hash value is stored. If additional instances of that hash are provided from the merge function, they are ignored until a new value is provided.

Generating candidate hashes

The first step, once a hash is passed to the rainbow table search routine, is to generate a series of candidate hashes. These hashes are specific to a password length, reduction function, and index. They are the result of regenerating the chains for each possible hash position within the chain (figure 10).

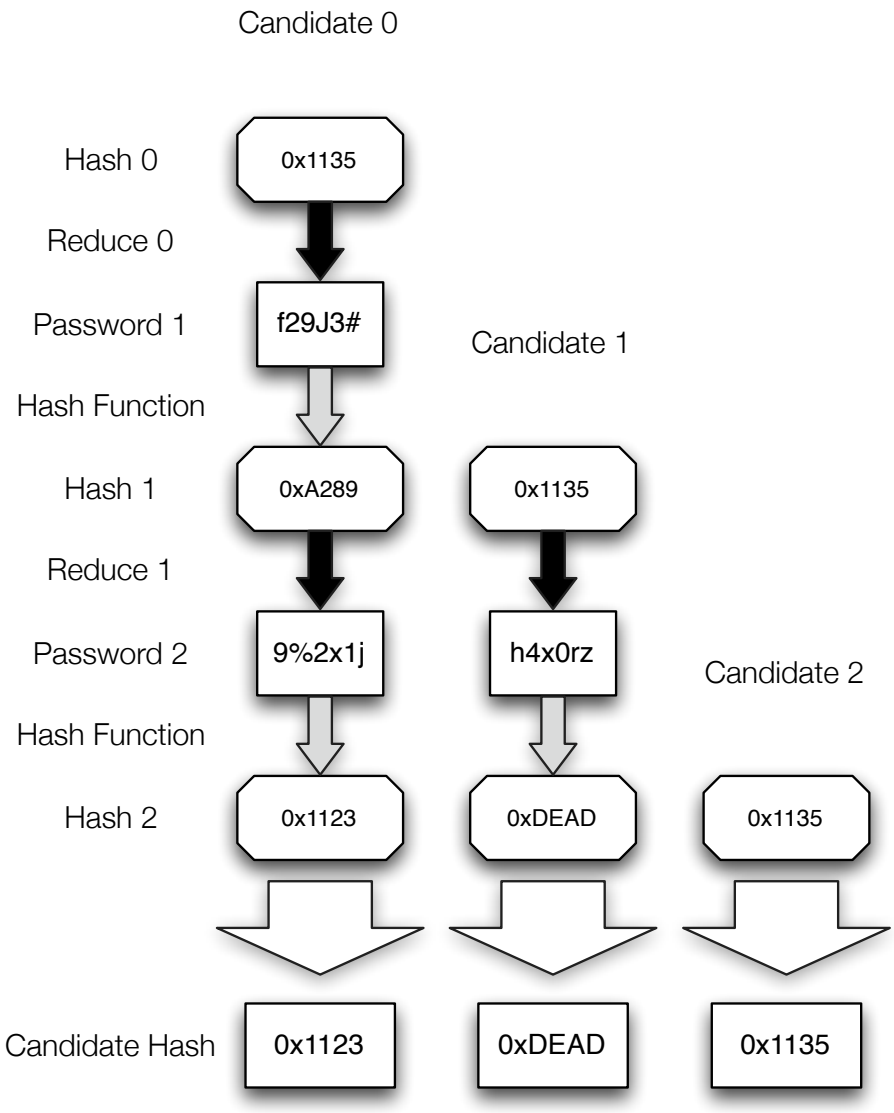

Figure 10: Rainbow table candidate hash generation

In figure 10, the candidate hashes are generated for 0x1135 starting at each of the chain positions, with the appropriate reduction function. Note that while candidate hashes 1 and 3 do not appear in the rainbow table (figure 7), 0xDEAD does. However, only one of the chains in the rainbow table ending in 0xDEAD has the hash in it. This is known as a "false alarm," and happens fairly often in large tables.

Candidate hash generation is the primary use of compute time in searching for a hash. As the chain must be regenerated for each of the possible hash positions, this generates a number of candidate hashes equal to the chain length. However, it also requires a very large number of steps to do this. The total number of steps required to calculate the candidate hashes is (0.5 * (chain length)^2). This places an upper bound on chain length, as the candidate hash generation time goes up with the square of chain length. However, as processing capability increases, longer chains can be used for new tables.

The candidate hash generation occurs on remote video cards. The network communication protocol for this process is seen in figure 11.

```
              Candidate hash data packets
[Hash type]\n
[Character set]\n
[Detailed character set info or blank]\n
[Chain length]\n
[Password length]\n
[Table index]\n
[16 bytes: hash]\n

Client returns, for each hash, in sorted order:
[16 bytes: hash]
```

Figure 11: Distributed candidate hash data packet

Once the candidate hashes have been generated, they are returned to the server. When the server has enough candidate hashes for a given table, it searches through the table to look for matching endpoints.

Searching the tables

After the candidate hashes have been generated, the previously generated tables are searched for matching end points. For all candidate hashes that match a chain endpoint, the initial password used to generate the chain is stored for the regeneration step. The searching of the tables effectively reduces the search scope from the entire password character space to the number of chains that have matching endpoints. In addition to valid matching chains, there are false alarms (matching endpoints that do not contain the password), and there is no guarantee that a given password/hash combination exists within a table. In figure 7, chains 1 & 2 would both be pulled for regeneration. Chain 2 contains the password, but chain 1 is simply a false alarm. It is also possible that if the table were perfected, chain 2 would no longer be present. In this case, despite the end hash being found, the password is not represented in the table and will not be found. Searching can only find passwords that are represented in the initial table.

Searching the tables is a very disk intensive task, and is performed on the central server. There are several options for searching the tables. The first, most commonly used option, is a binary search through the table. This appears to be a very fast option on paper, but forces the disk system to do a very large number of random seeks. Disks are relatively slow at this, and while the filesystem cache helps with subsequent searches on the same

table, this is a relatively slow method of searching a large table for large numbers of hashes. A binary search also requires somewhat complex logic at the end to handle the presence of multiple values of a hash in the file. While it can be effective on a perfected table, it is of less use for a full table.

Another option is to read linearly through the entire table. This makes far better use of the disk subsystem, as disks are significantly better at linear reads then they are at random seeks. However, when searching for a single set of candidate hashes, the hit rate is very low, and this wastes much time, as a very large file will not fit in memory, and will have to be pulled from disk each time.

The option used by IseCrack is to search linearly through the tables for a very large number of candidate hashes at once. The search code first loads as many candidate hashes as are available (up to a large limit, currently set at 1000) into memory. The hashes within these files are all in sorted order. Each candidate hash file is loaded, and an output file is opened to contain the chains to regenerate. The candidate files are then merged together to create a large sorted structure in memory. Each element contains the hash to search for, as well as the output file the chain information should be dumped to.

Once this large sorted structure is finished, the table file is opened, and read linearly. For each match, the chain information is appended to the appropriate output file. The search algorithm properly handles both multiple instances of a single hash in the table file and multiple instances of the same hash in the input chains. The end result is similar to a SQL JOIN statement, with each of the matching chains being present in each output file. At the completion of the table search, all matching chains are stored in files for regeneration and hash searching.

Chain regeneration and hash searching

The final step of the rainbow table process is to take the matching chains found in the table search step and regenerate each of them while looking for the specified hash. If the hash is found, the previous password is a valid reversal of the hash.

The chain regeneration can proceed in parallel, both for all hashes being searched for and for every chain within each set. As such, it is distributed to video cards through a network daemon.

```
          Chain regeneration data packets
[Hash type]\n
[Character set]\n
[Detailed character set info or blank]\n
[Chain length]\n
[Password length]\n
[Table index]\n
[16 bytes: hash to search for]
[Number of chains to regenerate]\n
For each chain to regenerate:
[16 bytes: initial passwords, null padded]

After searching, client returns:
[1 byte: 0 if failure, 1 if success]
If success:
[16 bytes: Password found, null padded]
```

Figure 12: Distributed chain regeneration data packet

The clients process the chains in parallel and return the result. If the hash is found, the corresponding row in the database is updated, and no further searching occurs against the hash. If the hash is not found, other tables and indexes are searched.

Brute Forcer

While the bulk of IseCrack is focused on rainbow table generation, the goal of the system is to be a rapid hash reversal system, able to quickly return passwords for provided hashes once the tables have been generated.

Passwords in a very small password space (5 characters or less, numeric passwords of 10 characters or less, lowercase only passwords of length 8 or less) are significantly slower to attack with rainbow tables than with a brute forcer. A fast brute forcer can return these passwords in seconds to minutes without ever having to touch a rainbow table. By running hashes through a brute forcer first, simple passwords are returned quickly. This accomplishes several things. First, as the goal of the system is to reverse hashes quickly, a simple password returned quickly accomplishes the task. Second, by filtering out the easy passwords, the significantly more compute-intensive rainbow tables can be reserved for the more complex passwords. By reducing the number of passwords that get passed to the rainbow tables, the rainbow tables can return complex passwords more rapidly. Finally, it is impractical to create rainbow tables for password scopes that can be represented in under 100 chains.

To allow for the greatest effective searching speed, the brute forcer takes large numbers of hashes and searches them in parallel (generate hash, check against all submitted hashes, generate next hash, check against all submitted hashes). While this slows the step rate through password space dramatically, it results in a very significant speedup in the total hash search rate (as it is searching many hashes in parallel). Because this system is designed for handling large numbers of hashes, this is a very effective tradeoff.

The IseCrack brute forcer is capable of handling up to 1000 NTLM hashes at once (a limit created by the 16384 bytes of shared memory per block; 1000 hashes at 16 bytes per hash takes up most of the space, with the character set taking up another 128 bytes), and is currently able to handle lengths through 8 characters (with search time being dependent on the character set used). The brute forcer is slower for a single hash than a brute forcer optimized for searching only single hash, but due to the number of hashes being searched in parallel, is significantly faster on a per-hash basis when multiple hashes are solved in a single pass.

Under typical expected use, the brute forcer will be used to test the full character set for lengths 1-5, uppercase and lowercase through length 8 (only testing length 6-8 as 1-5 have already been tested), and numeric through length 10. This will allow for the use of fewer distinct rainbow tables, and overall faster system performance on large numbers of hashes.

Utility functions and programs

During development and testing, it is important to verify that all functions are working as expected and that the data returned is correct. As the operation of the system is entirely dependent on hashes, reductions, and chains being correctly generated (for table generation, candidate hash generation, and final chain regeneration), it is vital to ensure that the tables and data are correct. Additionally, video cards are known for having a higher bit error rate than other devices. The actual rates are not available for public review, as they are under NDA, but it is important to verify that the cards are not generating errors in computation.

Additionally, the functions used for the verification code are independently written, using more standard ways of dealing with character arrays. This allows an easy check that the GPU code, dealing with little endian words, is generating the intended result. The verify code also uses the standard libssl MD4 hash function as opposed to the heavily optimized version used on the GPUs. This prevents a hash algorithm error from returning incorrect results.

The following utility programs have been written:

- generate_chain takes a provided password, chain length, and table index. It runs the specified number of steps, and outputs the final hash. Optionally, it will print the entire chain, with password/hash values at each step. This is useful to confirm chains are being properly generated, as well as to submit test cases that are verified as present in a table.
- test_reduction is used to verify the reduction function. It generates a number of chains, and at each step converts the password into a numerical index. This index in a large array is incremented. After a sufficient number of chains are generated, the password counts are output to observe the password distribution. If the reduction function is causing clustering (certain passwords represented significantly more often than others) or large areas of no passwords, this allows these behaviors to be detected and fixed.
- verify_table is given a table and the generate parameters for it (password length, chain length, index). It reads all the chains in the table and recalculates them on the CPU to confirm they are correct. Optionally, for large tables, verify_table can take a stride, only testing every N chains. This allows a rapid search through a table for

massive generate errors without having to compute the entire table (as this is significantly slower on the CPU than on a GPU).

Chapter 7: System Architecture and Design

IseCrack is a password cracking system, as opposed to a simple rainbow tables implementation. It also deals with password spaces that are, until recently, considered to be computationally infeasible to crack. As such, it is dealing with huge amounts of data from a large number of GPU compute nodes. These nodes should not require significant management once running – the server should be able to assign tasks to them as needed. Several of the design criteria and decisions considered:

- Despite the high computation speed of GPUs, the system will still take many GPU-years to fully compute the desired tables and will require many compute nodes feeding data into a central server or set of servers. The central server design needs to be able to handle this flow of data.
- While the system is generating data, end users should be able to submit password hashes and check them against the currently generated tables and brute force modules without needing to stop the table generation process.
- Idle GPUs are a waste of time and resources. The system architecture should be able to keep busy as many GPUs as are connected, either searching for hashes or generating new table data (unless all requested tables have been generated).
- Compute nodes may come online or go offline at any point and should be assigned work as they become available. There should be a way for a node to cleanly exit after completing it's current assigned work units. Also, a compute unit losing a network connection or shutting down mid-work unit should not cause any data to be permanently lost (except table part data). Any unfinished work units should be reclaimed and handed to a different compute node.

- With a full cluster providing data, the central server may be receiving 50-80Mbit of data from 50 or more clients. The server is designed to handle this, as long as the disk subsystem can keep up (which should be easy for any modern disk system).
- The data sizes within each table and overall will be very large - beyond the 4GB limit of a 32-bit memory space. A 64-bit OS is required to host the server.
- Because the final design goals were not provided, the system is being designed for maximum overall throughput with a large supply of password hashes. The overall system crack rate is the primary design criteria, as opposed to the time to crack an individual hash.

Of the tasks present, several can be pushed to any compute node easily, and several need to remain on the local machine or disk cluster.

- Table generation, candidate hash generation, chain regeneration, and brute forcing can be executed on any node.
- Table part merging and searching within a table for candidate hashes must be performed on the local system, as the data size is such that transferring it over the network is not feasible.

The final design for the system involves the creation of a small distributed computing project. Data is centrally stored on the server filesystem and managed through a MySQL database. The tasks that must be run locally are run locally, and the tasks that can be distributed across the network are handed out, in work chunks, by several network daemons running on the server. The network daemons interface to a MySQL backend, and allow the system administrator to change the priority and nature of workloads "on the fly" by changing

values in the database. If system requires additional nodes to handling the searching, the compute clients do not need to be restarted, but instead will change their workload as the server changes the work units handed out.

Compute nodes connect to the server, receive their work, perform the requested processing, and submit the results back to the server. If the server does not receive data back (due to the compute node crashing or going offline), the work unit will be reclaimed and provided to another system. The MySQL backend handles much of the data processing, and allows the network daemons to be dramatically simpler. Including support for additional hashes, character sets, and brute force modules is also simplified, as the network daemons do not need to be recoded. Adding the additional information into the database will allow the new information to be instantly passed out to clients. If a client does not support the workunit assigned to it, the client will deny the workunit and wait for a different one.

The system administrator can also specify priority of hashes to run through the system. While hashes of standard priority are handled in a FIFO queue, the administrator can specify that certain hashes are of a higher priority, and they will be processed before any other hashes waiting in the system.

Finally, a web interface allows the management and monitoring of the system. Hashes are added this way, and character set/hash types are added (though the compute clients must support the character set and hash type).

A weakness of the network system is that, if access to the compute network were gained, a malicious user could submit false data and corrupt the system. Possible defenses against this would be to randomly check submitted chains for correctness or submit work units to multiple nodes and compare results. As the system is designed to be operated on a secured

network, these are not currently implemented. Due to the bandwidth requirements for transfer of data, this system is unlikely to scale as-is to a standard distributed project, and so will not be accepting compute nodes from the internet at large. A future expansion would involve authenticating compute clients and confirming that the data returned was valid. An easy way to do this would be to check random chains from returned hashes, or to submit each work unit to multiple clients.

The system is hosted on 64-bit Linux servers. This allows dealing with very large files in a straightforward manner, as >4GB files can be accessed through memory mapping, and processes that need a large working memory space do not need to be PAE aware. The filesystem for the primary data stores is SGI's XFS. XFS supports the very large files and volumes needed (8 exabyte files, 16 exabyte volumes), has excellent performance on large files, and supports online defragmenting and resizing, which, if combined with a suitable RAID controller, would allow disk space to be purchased and added as needed, without requiring the host server to shut down.

Chapter 8: Ethical Considerations

Like many security-related projects, this project brings up ethical concerns. The primary concern voiced is that providing a "better password cracking system" will allow cybercriminals easier access to a user's passwords, with the subsequent problems that a compromised password causes, including the possibilities of identity theft, information theft, spam, and even financial loss (if the cracked password allows access to a company's sensitive information or an individual's bank account). A system that rapidly cracks passwords previously thought secure does certainly benefit those who are out to do harm with passwords, but that is not the only factor to be considered.

There are a variety of legal and ethical uses for password cracking systems. In addition to law enforcement, password cracking systems are commonly used by IT professionals in large and small organizations to test user-provided passwords and ensure they meet certain requirements. If the system administration staff is easily able to crack a password, any attacker would easily be able to do the same. Many organizations regularly test passwords with the currently available cracking software and require users to change easily compromised passwords.

The more important consideration is that none of the technologies used in this project exist alone, or are isolated to security researchers. This same knowledge and technology is available to both the defending side and the attacking side. The only difference is that the attacking side is frequently far more secretive about their tools and resources. As cybercrime is now a well-organized, profitable enterprise[10], tools and resources that improve the ability of criminals to gain access to machines for various purposes are worth money. While security researchers find many vulnerabilities, in some cases, the first indication a

vulnerability exists is a 0-day exploit in the wild, compromising systems and creating botnets while spreading to new hosts. There are auction sites for 0-day exploits[13], established pricing structures for botnet attacks (spam and denial of service are the most common)[14], and well established chains for trafficking of identities and information.

There are individuals in the cybercriminal realm who have devoted time and resources to password cracking. The application of video cards to rainbow tables is a clear match for those familiar with both, and while the skills needed to implement a solution are beyond what many involved in cybercrime have, it only takes one or two skilled individuals to write the code that can then be utilized by others (as is currently the case for much of the trojan and botnet code). Alternately, an individual or group of individuals could build their own cracking cluster, and sell "cracks" - taking hashes and reversing them. The total resources needed are well within the reach of any moderately funded cybercrime organization, as well as within the reach of a decently funded individual. This ignores the fairly likely possibility of obtaining all the hardware needed with stolen and fraudulent credit card numbers.

Given all this, it is reasonable to assume that there are individuals in the cybercrime world actively pursuing fast password cracking with GPUs. A GPU cluster could be easily applied to a wide variety of cracking, including wireless networks. The presence of such a system would be difficult to discover, as it would likely remain hidden, with just the output being released to others, and most likely for pay.

Building a proof of concept system with the technology is, then, simply bringing awareness to existing technologies that can be combined and used. Additionally, defending against rainbow tables is very easy, but requires the system programmers be aware of the

threat and the capabilities of current hardware in attacking hash-based password systems. IseCrack, then, serves as a public demonstration of the power of GPU based password cracking systems, and a warning that non-salted passwords are not secure in the lengths most commonly used.

Chapter 9: Performance

The goal of a rainbow tables implementation on graphics cards is high performance password cracking. If GPUs are not able to significantly accelerate password cracking, there is no point to using them.

Fortunately, they do provide very significant speedups against CPU-based rainbow table implementations. As the current reference implementation of rainbow tables is Rainbow Crack, this is used as a benchmark for comparison.

```
rgraves@isecrack-server:~/rainbowcrack-1.2-src/src$ cat /proc/cpuinfo |
grep CPU | tail –n 1
model name      : Intel(R) Xeon(R) CPU           E5345  @ 2.33GHz

rgraves@isecrack-server:~/rainbowcrack-1.2-src/src$ ./rtgen ntlm all 8 8
0 -bench
ntlm hash speed: 3623188 / s
ntlm step speed: 2173913 / s
```

Figure 13: RainbowCrack performance

Rainbow Crack, on a Core 2 based Xeon at 2.33ghz, completes 2.1M links per second per core, for a total system generation rate of 8.4M links per second on the quad core Xeon.

There exist GPU accelerated versions of the rainbow crack generator, but they are all either unable to be downloaded (invalid links), or are Windows-only and do not support NTLM. However, it appears they are able to run much better generate rates of 70-80M links per second. This is an improvement, but speeding the generate does not solve the problem of chain storage and crack speed with longer chain lengths. All these products appear to generate chains of length 10,000 (the standard Rainbow Crack length), and they are implementing the slow Rainbow Crack reduction function.

Free Rainbow Tables is currently generating chains at around 8000 chains per second, which translates to (8000 * 10000) = 80M links per second. However, some of their computing resources are used for searching and cracking as well.

IseCrack is substantially faster than any of these, both due to algorithmic improvements and to the GPUs. To verify the algorithmic improvements and compare performance directly, a CPU version was created.

```
Starting kernel: NTLM/all Length 7, Index 0, 100 chains of length
100000
Kernel Time for 100 chains: 1110.000 ms
Step rate: 9.01 M/s
Writing results to network.
Writing 100 chains to network completed.
```

Figure 14: IseCrack CPU performance

IseCrack's algorithm, run on a CPU, completes 9M links per second per core, for a total system generation rate of 36M links per second.

On a single nVidia 8800GTX OC (128 stream processors, shader rate of 1.46ghz), IseCrack is able to generate chains at a stepping rate of 410M links per second.

```
Starting kernel: NTLM/all Length 7, Index 2, 100000 chains of length
100000
Kernel Time for 100000 chains: 24503.080 ms
Step rate: 408.11 M/s
Writing results to network.
Writing 100000 chains to network completed.
```

Figure 15: IseCrack GPU performance

A GTX260 (192 stream processor edition) was able to generate at over 450M links per second. Estimated performance on a 9800GX2 (two 9800 cards, bound together into a single package) is around 900M links per second. These rates are valid for all the chain stepping code, including table generation, candidate hash generation, and chain regeneration. Chain regeneration is slightly slower due to having to check for hash matches, but this does

not affect overall system performance, as the chain regeneration step involves the least work of any of the GPU accelerated steps.

For brute forcers, ElcomSoft's NTLM brute forcer claims to run at 365M passwords per second[4]. BarsWF NTLM cracker runs at 500M passwords per second on a 9800GTX, which is a significantly faster card than the 8800GTX used for testing.

The brute forcer module in IseCrack is not optimized for single password speed. Instead, it is optimized for total throughput. When running with 1000 hashes to check, depending on password length, it achieves a total hash check rate of 20-25B hashes per second. Its single hash performance is adequate, at around 400M steps per second, but adding additional hashes to check rapidly raises performance well beyond what any other brute forcers are currently achieving.

Chapter 10: Results & Conclusions

As a password cracking system, IseCrack is successful at generating performance not previously seen in password cracking systems. Existing rainbow implementations have not fully embraced GPU acceleration, which limits their ability to represent large password spaces in reasonable disk space. IseCrack is, to my knowledge, the first fully GPU accelerated rainbow table implementation.

The limits of current rainbow tables are being used as test cases for IseCrack. Neither Free Rainbow Tables nor OphCrack support NTLM hashes with a full character set beyond 6 characters. IseCrack has used that for testing and verification as a trivial case. Extending the full character set out to 8 characters is relatively simple, and 9 characters are feasible with sufficient GPU power available. Substantially longer password attacks can easily be made on smaller character sets within very feasible amounts of time.

IseCrack clearly demonstrates that applying GPUs to rainbow tables allows a very significant speedup (200x or more) against existing solutions, and it allows for the attack of passwords that had previously been considered secure.

In light of the performance numbers seen, it would not be advisable to use anything under a 12 character password with all character classes (upper, lower, numeric, symbol) represented, and for passwords using fewer character classes, longer passwords are important. It also highlights the importance of salting passwords, as a salted password is not nearly as vulnerable to a rainbow table as an unsalted password.

Chapter 11: Future Work

IseCrack has significant potential for future expansion. The current implementation, while functional, is a proof of concept running on a small number of video cards. The system, to meet the intended goals, needs significantly more computational power. This requires a central server with significantly more storage, and the processing and memory capacity to handle the flow of data.

This also requires significantly greater numbers of video cards. One possible option for this is a cluster of GPUs, either nVidia Tesla servers or separate systems. Alternately, time could be obtained on an existing GPU cluster. Another option is to turn the project into a distributed computing project, allowing contributing users to crack hashes with the system in exchange for GPU compute time. This is a preferred option if it can be done, as the power is functionally free for the project, and a significantly larger number of GPUs can be obtained for computation.

Additional work will involve supporting more hash algorithms. Other commonly used algorithms that need additional work are LM (LanMan hashes, DES based), MD5, SHA1, and new algorithms being produced. IseCrack is a solid framework to extend to these hash types.

Bibliography

Philippe Oechslin. "Making a Faster Crytanalytical Time-Memory Trade-Off" Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings. Lecture Notes in Computer Science 2729 Springer 2003, ISBN 3-540-40674-3

[1] "AMD Stream Processor First to Break 1 Teraflop Barrier." AMD. June 16 2008. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_15434~126593,00.html. [Accessed Nov 20 2008]

[2] Schmid, Patrick. "Tom's Hardware's 2007 CPU Charts." Tom's Hardware. July 16 2007. http://www.tomshardware.com/reviews/cpu-charts-2007,1644-36.html [Accessed Nov 23 2008]

[3] "Project Rainbowcrack." http://www.antsight.com/zsl/rainbowcrack/ [Acessed Nov 2 2008]

[4] "ElcomSoft Distributed Password Recovery : High-performance distributed password recovery with NVIDIA GPU acceleration." http://www.elcomsoft.com/edpr.html [Accessed Nov 28 2008]

[5] "Free Rainbow Tables." http://www.freerainbowtables.com/ [Accessed Nov 30 2008]

[6] "OphCrack." http://ophcrack.sourceforge.net/ [Accessed Nov 10 2008]

[7] "World Fastest MD5 cracker BarsWF." http://3.14.by/en/md5 [Accessed Nov 20 2008]

[8] "Vote for the distributed project name." http://3.14.by/forum/viewtopic.php?f=8&t=20 [Accessed Nov 28 2008]

[9] "Pixel Shaders." Nvidia.com. http://www.nvidia.com/object/feature_pixelshader.html [Accessed Nov 20 2008]

[10] "How Cybercrime Became a Booming Business." Best Security Tips. http://www.bestsecuritytips.com/news+article.storyid+614.htm [Accessed Nov 18 2008]

[11] “CUDA for GPU Computing.” nVidia.com http://news.developer.nvidia.com/2007/02/cuda_for_gpu_co.html [Accessed Nov 5 2008]

[12] “rtgen GPU.” http://www.freerainbowtables.com/phpBB3/viewtopic.php?f=5&t=611 [Accessed Nov 28 2008]

[13] “WabaSabiLabi FAQ.” WabiSabiLabi. http://www.wslabi.com/wabisabilabi/faq.do? [Accessed Nov 29 2008]

[14] Hruska, Joel. “Study: Storm botnet brought in daily profits of up to $9,500.” ArsTechnica. http://arstechnica.com/news.ars/post/20081110-study-storm-botnet-brought-in-daily-profits-of-up-to-9500.html