Generating Super Magic Hashes
A Parallel Approach
Outline for this presentation

- Recap of Magic Hashes and Generation
- Progress (thus far)
- Results
- Discourse
Magic Hashes
(and Type Juggling)

Interactive shell

```php
php > var_dump("0e229758" == "0e000000");
bool(true)
php > //Oops.. ;)
php >
```

Type-Juggling
Magic Hashes

- Specific hashes used to exploit Type Juggling attacks in PHP
- Can be used to detect vulnerabilities in authentication flows

```php
php > $user_input = md5(2406108708);
php > $test = 0;
php > if ($user_input == $test) { echo "EQ" ;} else { echo "Not EQ" ;}
EQ
```

In PHP two strings matching the regular expression `0+e[0-9]+` compared with `==` returns `true`:

```
'0e1' == '00e2'== '0e1337' == '0'
```

Indeed all these strings are equal to 0 in scientific notation.
Vulnerability Detection

1. Find 2 Magic Hashes to work as passwords
   for e.g. - ‘lowercasegzmqqmx’ and ‘lowercasifdvqkfr’
2. Register for a website with Password 1
3. Attempt to sign in with Password 2
4. If sign-in is successful, the system uses the specified algorithm
   and ‘==’ to compare them.
5. Vulnerability detected
Generating Magic Hashes - SHA1

- SHA1 Digest - 160bits or 40 HEX characters
- Total # of Hashes - $2^{160}$
- Total # of Rounds per hash - 80
- Each round generates a subset of the digest which is input to the next round
- Ex: - 0e12149120354415335220758399492713921588
- Ex: - d4ee942416a6e4aad41941c1a6a0f92ac097661b
Generating Magic Hashes - SHA1

- Benchmark for 10 million hashes generation @ 292ns/op
- Our requirement is to get ~2.4 trillion hashes to get >50%
~8 days to have a 50% chance of getting a Magic Hash
Parallelising the Generation - Progress
Goals

Idealised Algorithm

• Use 'N' processors, each generating a password.

• Pairs of processors generate code sections, divided into small and large chunks. They exchange the generated strings for hashing.

• The exchanged strings are hashed, ideally using shared memory for the 80 rounds. OpenMP with shared memory can be used for parallelisation since each round depends on the previous one.

• After generating the magic hash, broadcast it to all processors to conclude the algorithm.
Goals

Idealised Algorithm

• Use 'N' processors, each generating a password.

• Pairs of processors generate code sections, divided into small and large chunks. They exchange the generated strings for hashing. EXTREMELY INEFFICIENT

• The exchanged strings are hashed, ideally using shared memory for the 80 rounds. OpenMP with shared memory can be used for parallelization since each round depends on the previous one. IMPOSSIBLE FOR SHA1 ALGORITHM

• After generating the magic hash, broadcast it to all processors to conclude the algorithm.
Proposal for Parallelisation

How do we make it faster?

• Level 1 - Run code on a single processor with 'N' cores.

• Level 2 - Split Generation and Hashing across 'N' processors

• Level 3 - Split Generation between pairs of processors - Little/Big Endian style

• Level 4 - All the above across 'N' processors with each subprocess multithreaded/multicored

• Level 5 - All of the above but now on N nodes/machines
Proposal for Parallelisation

How do we make it faster?

• Level 1 - Run code on a single processor with ’N’ cores. ✓

• Level 2 — Split Generation and Hashing among ’N’ processors ❌

• Level 3 — Split Generation between pairs of processors — Little/Big Endian style ❌

• Level 4 — All the above amongst ’N’ processors with each subprocess multithreaded/multicored ❌

• Level 5 - All of the above but now on N nodes/machines ✓

• Level 6 - Micro-optimise sections of code instead of parallelising ✓
Progress (?)
Randomness Hurdle
The problem with random generation

- Randomly generated strings generate random results
- Random results = no distinction between Dependent v/s Control variables
- No distinction = no scientific conclusion
- Also, running time in hours constrains number of experiments

SHA1, Password Len = 16

Running Time (in hours)

<table>
<thead>
<tr>
<th>Number of PEs</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>16</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>3.5</td>
<td>7</td>
<td>10.5</td>
<td>14</td>
<td>3.5</td>
</tr>
</tbody>
</table>
Randomness Hurdle
The Solution

- Use a known Magic Hash
- Generate Upper bounds from this hash
- Uniform comparisons across each PE
- Runtime can be reduced to seconds or minutes based on architecture and hardware capabilities. (We control this)
Recursive Doubling Example
with 4 PEs and 1000 max ops

Start

PE #1

PE #2

PE #3

PE #4

Preselected Magic Hash - "2000"

Calculated Max Ops - 1000
Generate passwords starting from 1000 to 1250

Start ➔ PE #1 ➔ PE #2 ➔ PE #3 ➔ PE #4

- Data per processor - 1000 / 4
- Generate next Base Password from "1000"
- Base Password - "1250"
- Calculated Max Ops - 1000
Start

Generate passwords starting from 1000 to 1250

Generate passwords starting from 1250 to 1500

Generate passwords starting from 1500 to 1750

Generate passwords starting from 1750 to 2000

PE #1

PE #2

PE #3

PE #4
Overview of Algorithm

1. Initialise MPI Processors
2. Generate Random String
3. Split String and broadcast/recv
4. Generate SHA1 Hash
5. Check for magic hash
6. Continue, within bounds, until found

```c
int main(int argc, char *argv[])
{
    // To ensure consistent results – 1 billion ops
    const unsigned long int MAX_OPS = 1'000'000'000;
    int total_proc = init_MPI_comm();
    int curr_proc = init_my_MPI();

    // For each processor, only done once
    const int UPPER_LIMIT_BOUND = MAX_OPS / total_proc;
    std::string pwd = generate_random_password();
    pwd = split_pwd(curr_proc, UPPER_LIMIT_BOUND, pwd);

    // Loop until upper limit is reached
    while (true)
    {
        if (is_magic_hash(pwd))
        {
            notify_all_processors();
            return 1;
        }
        else if (within_upper_bound(pwd, UPPER_LIMIT_BOUND))
        {
            increment_char();
        }
        else
        {
            // Current processor has hit its upper limit
            break
        }
    }
    return 0;
}
```
Overview of Algorithm

Password Splitting

- Every $\log n$ processor in MPI_COMM_WORLD generates the next base password.

- Broadcast to next $\log n$ processors so they can start generating and hashing.

- Every other processor in $[\log n + 1, 2\log n]$ receives base password to compute.

```cpp
std::string split_pwd(int curr_proc, int UPPER_LIMIT_BOUND, std::string pwd) {
    // If processor is power of 2, it generates the next set of base pwd
    if ((curr_proc > 0) && ((curr_proc & (curr_proc - 1)) == 0)) {
        // Recursively double the generated password
        // Send the generated password to all processors with ranks below
        int bounding_processor = curr_proc * 2 - 1;
        for (int i = curr_proc + 1; i <= bounding_processor; i++) {
            MPI_Send(splitPwd, split_pwd.size(), MPI_BYTE, i, 0, MPI_COMM_WORLD);
        }
        int number_of_places_away = curr_proc + UPPER_LIMIT_BOUND;
        std::string splitPwd = incrementStringFor(pwd, number_of_places_away);
    } else {
        std::string newPwd;
        MPI_Recv(&newPwd,_pwd_LEN, MPI_BYTE, nearestPowerOfTwo(curr_proc), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```
Overview of Algorithm

Character ‘Incrementer’

- Increment each character until the boundary of alphanumeric characters and reset.
- Two For Loops means potential for shared memory optimisation.
Overview of Algorithm

Magic Hash Checker

- Generates and checks hash
- Magic Hashes only need to start with any amount of 0s and one count of ‘e’
- Fastest way to match string patterns is regex

```cpp
bool is_magic_hash(std::string pwd)
{
    std::string hash = generate_hash(pwd);
    std::regex const regExp{"^\^0+e\d*$"};
    std::smatch matched_arr;
    if (std::regex_match(hash, matched_arr, regExp))
    {
        return true
    }
    else
    {
        return false
    }
}
```
Where are my cycles?

• The 3 Big Functions for our flow

1. Password Generator
2. SHA1 Hash Generator
3. Increment Character

• Password Generation happens only once per node/PE
Micro-Optimisation #1
Password Generation

- Use array of bytes instead of strings for passwords
- Requires less allocations if not using a heavy class like std::string
- Less allocations = less wasted cycles = faster runtimes

![Bar chart showing performance comparison between PasswordGeneration, SHA1Generation, and IncrementCharacter. The chart indicates that IncrementCharacter is 2.2 times slower than PasswordGeneration and 1.2 times faster than SHA1Generation. The lower ratio indicates faster performance.](image)
Micro-Optimisation # 2
SHA1 Hash Generation

• Use Intel’s SHA function that is already baked into most modern CPUs
• Uses CPU Instructions to compute hashes efficiently
• (Don’t reinvent the wheel)
• Doesn’t work for SHA224 (Couldn’t find one)
Micro-Optimisation # 3

String <-> Byte conversion

- Replace string comparisons everywhere with byte comparisons
- No more string conversions or manipulations means CPU registers are better utilised
- Huge gains!
Micro-Optimisation # 4

OpenMP Shared Memory

- Use OpenMP for Password Generation
- Pragma OMP Reduction clause works very well with FOR loops
Micro-Optimisation # 4

OpenMP Shared Memory

• Use OpenMP for Password Generation

• Pragma OMP Reduction clause works very well with FOR loops

• Tiny gains but gains nonetheless

![Bar chart showing performance comparison between PasswordGeneration, SHA1Generation, and IncrementCharacter]
Results
Timeline
Previous progress

• Run a sequential version of the final algorithm on device
  ~11 hours

• Run the sequential version on cluster (without any optimisation)
  ~13 hours

• Split the string generation and hash verifier pieces of code (no other forms of
  communication between processors)
  ~9 hours

• Split the string generation into chunks (communication between pairs of
  processors)
  ~16 hours
Timeline

Current Results

• Run a sequential version of the final algorithm on device
  [30 - 300] seconds

• Run the sequential version on cluster (without any optimisation)
  ~13 hours

• Split the string generation and hash verifier pieces of code (no other forms of
  communication between processors)
  ~9 hours

• Split the string generation into chunks (communication between pairs of processors)
  ~16 hours

• Run the final algorithm on clusters
  [10 - 150] seconds
### Results - Running Time

SHA1, Password Length - 12

<table>
<thead>
<tr>
<th>PEs</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30.7</td>
</tr>
<tr>
<td>2</td>
<td>18.8</td>
</tr>
<tr>
<td>4</td>
<td>14.7</td>
</tr>
<tr>
<td>16</td>
<td>10.9</td>
</tr>
<tr>
<td>32</td>
<td>9.7</td>
</tr>
<tr>
<td>64</td>
<td>11.2</td>
</tr>
<tr>
<td>128</td>
<td>10.4</td>
</tr>
</tbody>
</table>

Graph showing running time in seconds against the number of processors for SHA1 with a password length of 12.
## Results - Running Time

### SHA1, Password Length - 40

<table>
<thead>
<tr>
<th>PEs</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.4</td>
</tr>
<tr>
<td>2</td>
<td>19.5</td>
</tr>
<tr>
<td>4</td>
<td>15.2</td>
</tr>
<tr>
<td>16</td>
<td>9.9</td>
</tr>
<tr>
<td>32</td>
<td>8.9</td>
</tr>
<tr>
<td>64</td>
<td>10.5</td>
</tr>
<tr>
<td>128</td>
<td>10.2</td>
</tr>
</tbody>
</table>

![Graph showing running time vs number of processors](image-url)
Results - Running Time

<table>
<thead>
<tr>
<th>PEs</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>312.9</td>
</tr>
<tr>
<td>2</td>
<td>257.3</td>
</tr>
<tr>
<td>4</td>
<td>266.5</td>
</tr>
<tr>
<td>16</td>
<td>187.2</td>
</tr>
<tr>
<td>32</td>
<td>161.1</td>
</tr>
<tr>
<td>64</td>
<td>157.2</td>
</tr>
<tr>
<td>128</td>
<td>178.7</td>
</tr>
</tbody>
</table>

SHA224, Password Length - 40
Results - Speedup

Amdahl’s Law

<table>
<thead>
<tr>
<th>PEs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.32</td>
</tr>
<tr>
<td>16</td>
<td>1.72</td>
</tr>
<tr>
<td>32</td>
<td>1.93</td>
</tr>
<tr>
<td>64</td>
<td>1.67</td>
</tr>
<tr>
<td>128</td>
<td>1.81</td>
</tr>
</tbody>
</table>

SHA1, Password Length - 12

Number of Processors

Speedup

1 2 4 16 32 64 128
1 1.32 1.72 1.93 1.67 1.81
## Results - Speedup

### Amdahl’s Law

<table>
<thead>
<tr>
<th>PEs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.28</td>
</tr>
<tr>
<td>16</td>
<td>1.96</td>
</tr>
<tr>
<td>32</td>
<td>2.19</td>
</tr>
<tr>
<td>64</td>
<td>1.86</td>
</tr>
<tr>
<td>128</td>
<td>1.92</td>
</tr>
</tbody>
</table>

SHA1, Password Length - 40

Number of Processors

![Graph showing speedup vs number of processors](image-url)
Results - Speedup

Amdahl’s Law

<table>
<thead>
<tr>
<th>PEs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.33</td>
</tr>
<tr>
<td>16</td>
<td>1.49</td>
</tr>
<tr>
<td>32</td>
<td>1.59</td>
</tr>
<tr>
<td>64</td>
<td>1.69</td>
</tr>
<tr>
<td>128</td>
<td>1.68</td>
</tr>
</tbody>
</table>

SHA224, Password Length - 40

Graph showing speedup vs. number of processors.
Discourse
Takeaway #1
Parallelisation is not always a silver bullet

- The initial algorithm's approach of splitting into chunks and exchanging communication resulted in wasted operations and idle cycles.
- Even with batching results, idle time was high.
- Long compile times meant not enough time to run diverse experiments.

Source: xkcd
Takeaway #2
Overengineering = Spaghetti Code

• The attempt to cleverly avoid idle time ended up introducing additional idle time in unintended ways.

• Over-complicating a straightforward algorithm inevitably leads to the development of convoluted and tangled spaghetti code.

• K.I.S.S prevails.

Source: xkcd
Takeaway #3
More Cores, Same Problems

- A subpar speedup was observed when utilising a maximum of 128 cores, despite minimal communication between the processors.

- The introduction of threads resulted in the emergence of synchronisation issues.

- Gated by SHA1 hash generation not being ‘parallelisable’.

Source: xkcd
Extension #1

GPUs

- GPUs exhibit hash rates that are 20 times greater than CPUs when it comes to generating hashes.
- CUDA Cores further simplify and optimise the hashing process, out of the box (OOTB).

Source: Hashcat
Extension #2
OpenMP

• Pragma OMP Reduction only optimises one variable and only supports few basic ops

• By virtualising both password generation and password hashing, the occurrence of idle or no-op cycles is further minimised or even eliminated.

Source: ResearchGate
Questions?

and Thank You
References

• Super Magic Hashes
• Chick3nMan and Spaze F0rze - Twitter
• PHP Magic Hashes
• SHA1 - Auth.0
• SHA-1 Collision
• SHA-1 CPU Extensions - Intel
• CCR Batching and Open MPI reference
• CCR Batch Jobs and Clusters
• ResearchGate - OpenMP