Password Cracking in Parallel

CMP202 JAKE BRETHERTON 1800231

The Problem:

Find an unknown password from a given hash.

Passwords are stored in a hashed format, meaning if there is a leak of a website's databases raw passwords will not be visible.

A password cracker will take a hash, and by brute force try possible passwords until one hashes into the provided hash.

(Actual Password)	Leaked Hash
Pass1	487314782

Guess	Hash	Same As Leak?
Pass0	348468795	No
Pass1	487314782	Yes

Found!

Simplifications

This is a toy model:

- The hashes will not be salted
- The hash algorithm used will not be cryptographically secure
  - It the principle is the same but the program will run significantly faster
  - C++ has a built in hash function for dictionary keys
  - Ethically, this will not be usable on actual leaks

Breaking down the problem

Crack the password:

- Generate Password guesses
- Hash the Password guess
- Compare generated hash to provided hash
- Return found password

Note on generating password guesses

There are 95 commonly used Unicode characters in passwords

- a-z
- A-Z
- 0-9
- !"# $%&'()*+,-./:;<=>?@[\]^_`{|}~
- Complete sequential block of characters from space to tilde, ' ' to '~', char 32 to 126

Almost every new guess will be the previous guess, with the last character incremented

Therefore 98.9%* of generation is taking the last character and incrementing it by 1

*(1 – 1/96)

```
aFeg
aFeh
aFei
aFej
aFek
aFel
aFem
aFen
aFeo
aFep
aFeq
aFer
aFes
aFet
aFeu
```

Breaking down generation

Expensive algorithm to increment the password root

- Can handle if last character of root is largest char
- Can handle adding new character to root if whole root is largest char

Cheap algorithm to increment the last character

- Increment char and store it along with the root
- Signal to expensive algorithm if char is largest char

98.9% of generations use only cheap algorithm

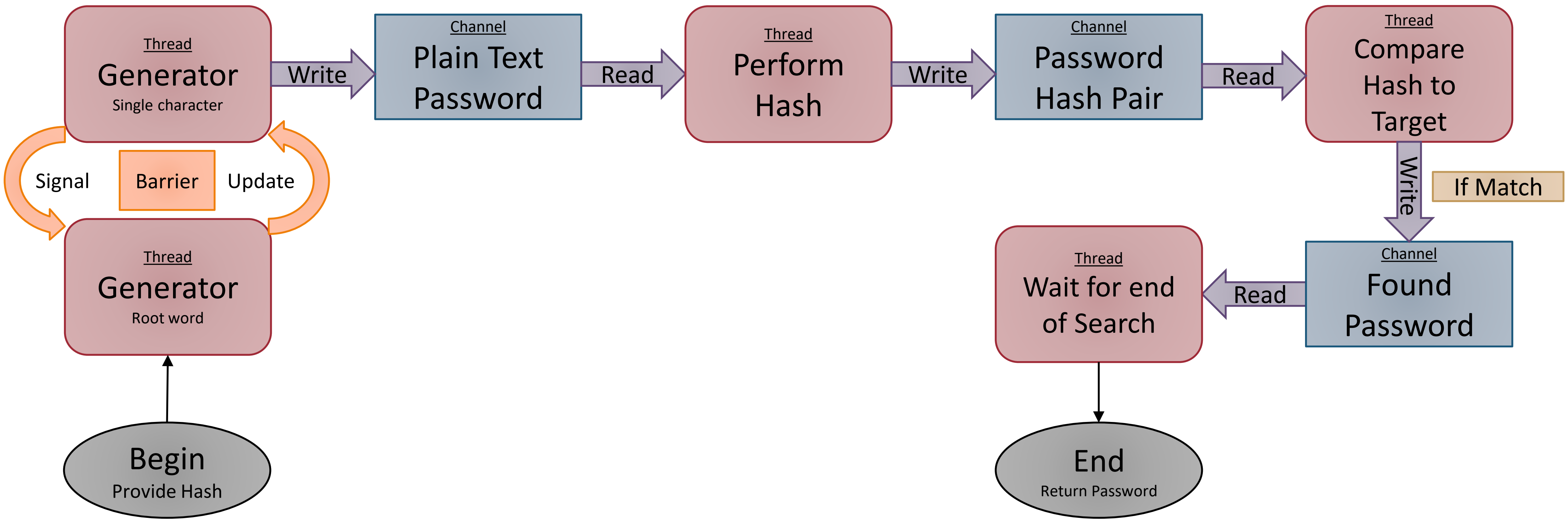
aFe-	-p	aFep
Root	Last character	Password

Breaking down the problem

Crack the password:

- Generate Password guesses
  - Increment password root
  - Increment the last character
- Hash the Password guess
- Compare generated hash to provided hash
- Return found password

Model



Thread
Generator
Single character
Write
Channel
Plain Text Password
Read
Thread
Perform Hash
Write
Channel
Password Hash Pair
Read
Thread
Compare Hash to Target
Signal
Barrier
Update
Write
If Match
Thread
Generator
Root word
Thread
Wait for end of Search
Read
Channel
Found Password
Begin
Provide Hash
End
Return Password

Thread managing tools

Need:

- Barrier
  - Suspends threads that reach barrier until specified number of threads reach barrier, at which point all are awoken
- Channel
  - Pipe to pass data between threads
  - Reading threads suspend until data is available, writing threads suspend until there is room in the channel
  - These require a semaphore
- Semaphore
  - Signal a count between threads
  - If count > 0, threads waiting for signal will decrement count and continue
  - If count = 0, threads waiting for signal will suspend until count > 0, when they will wake up
  - A thread signalling the semaphore will increment the count and wake up a sleeping thread waiting on the semaphore

None of these are available in the current version of C++ so they had to be written

Barrier

Arrival to the barrier is protected by a mutex lock, as it requires modifying the state of the barrier to increase the number of counted arrived threads (and possibly unlocking the barrier. There is an inherent race condition to arriving.

If not all threads have arrived at the barrier then the thread is suspended and the mutex lock automatically released.

```
void Barrier::ArriveAndWait() // suspends threads untill all have reached this point
{
    std::unique_lock<std::mutex> lock(mtx); // RAII mutex lock
    int useNumber = barrierUseCount; //  keep track of the current use of barrier (used to prevent spurious wakeup while being reusable)
    if (count >= limit || !enabled) { // if the count of waiting threads reaches the barrier limit, open
        barrierUseCount++; // next use of barrier
        count = 0; // reset count
        cv.notify_all();
    }
    else { // else increment count and set thread waiting
        count++;
        cv.wait(lock, [this, useNumber] {return useNumber < barrierUseCount; }); // prevent spurious wakeup, stop waiting if this barrier use has opened
    }
}
```

Semaphore

A mutex lock is acquired as signalling and waiting on a semaphore both alter the pool count and therefore represent a race condition.

If the thread suspends waiting for a signal then the mutex it automatically released until woken up (when it reacquired)

```
// increment pool count, wake up a sleeping thread on wait
void Semaphore::Signal()
{
    { // mutex scope
        unique_lock<mutex> lock(poolMutex); // aquire mutex lock as required by condition variable, RAII
        poolCount++;

    }
    cv.notify_one(); // we have only added one to the pool count so we only need to notify one thread to wake up and deal with it
}
```

```
// suspend thread until pool count is greater than 0, decrement pool count
void Semaphore::Wait()
{
    unique_lock<mutex> lock(poolMutex); // aquire mutex lock as required by condition variable, RAII
    cv.wait(lock, [this] {return poolCount > 0;});  // suspend thread if pool is empty until notified and pool is no longer empty
                                                    // this automatically releases the lock while thread is suspended
    poolCount--;
    return;
}
```

Channel

Requires two semaphores, one to hold the number of elements in the channel, another to hold the number of empty spaces in the channel.

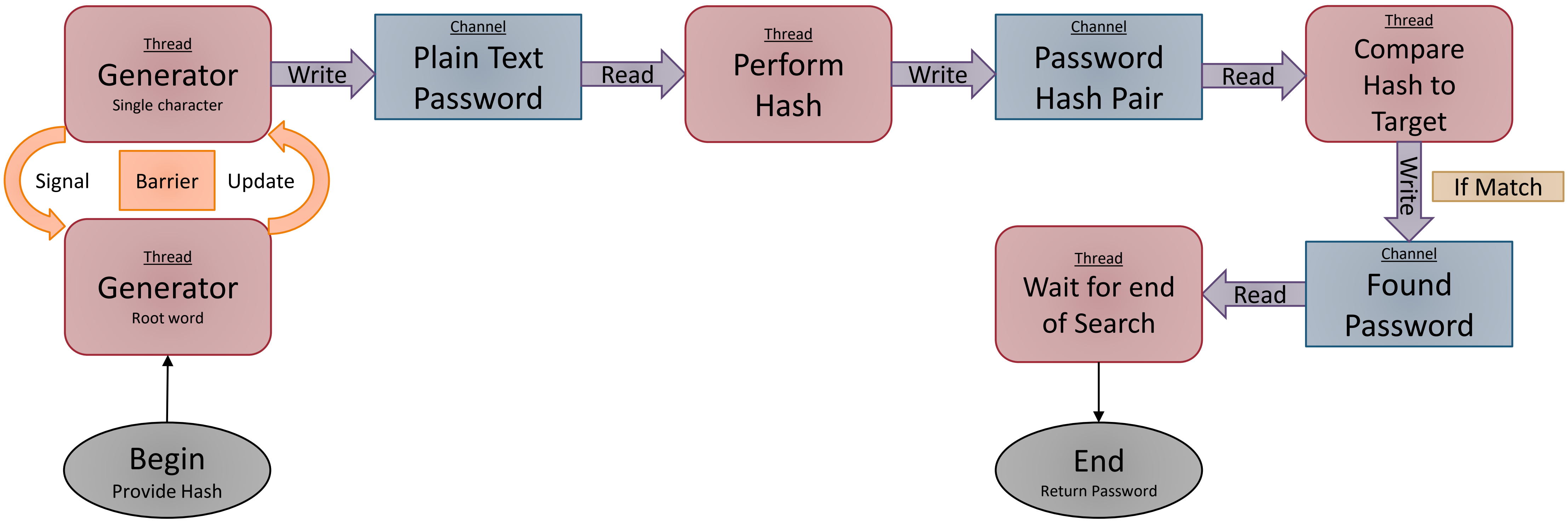
A mutex lock is needed to write to the channel, as well as read from it as reading removes the item read.

```
template<class T>
void Channel<T>::Write(T data)
{
    emptySem.Wait(); // block and suspend unless there is room to write into the buffer
    { // RAII scope
        lock_guard<mutex> lock(mtx);
        buffer.push_front(data);
    }
    sem.Signal();

}
```

```
template<class T>
T Channel<T>::Read()
{
    sem.Wait(); // block and suspend unless there are available elements in the buffer
    lock_guard<mutex> lock(mtx); // RAII mutex for readind and altering the buffer
    if (!enabled) return T{}; // if channel has been decommissioned, unblock but prevent reading empty buffer
    T item = buffer.back();
    buffer.pop_back(); // remove read item from buffer
    emptySem.Signal(); // signal that there is now extra room in the buffer
    return item;
}
```

Model


Thread
Generator
Single character
Write
Channel
Plain Text Password
Read
Thread
Perform Hash
Write
Channel
Password Hash Pair
Read
Thread
Compare Hash to Target
Signal
Barrier
Update
Thread
Generator
Root word
Write
If Match
Thread
Wait for end of Search
Read
Channel
Found Password
Begin
Provide Hash
End
Return Password

Generator threads

Last Character Thread

Take root and add character to it, write this to channel and increment character

When exhausted characters, arrive and wait to signal the root should be updated

Arrive and wait for root password to be updated

Root Password Thread

Arrive and wait for generation to finish

Update the root password

Once finished arrive and wait to signal generation may begin again

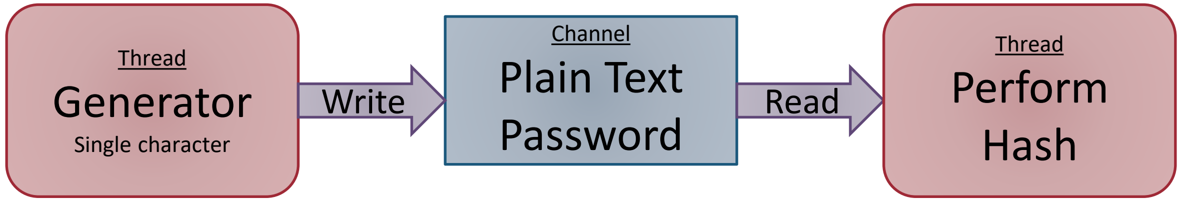
Generation to hashing

Hashing thread initially tries to read the password plaintext channel

This channel is empty so the thread suspends

The last character generator thread writes a string to the channel

This causes the hash thread to wake up and hash the password



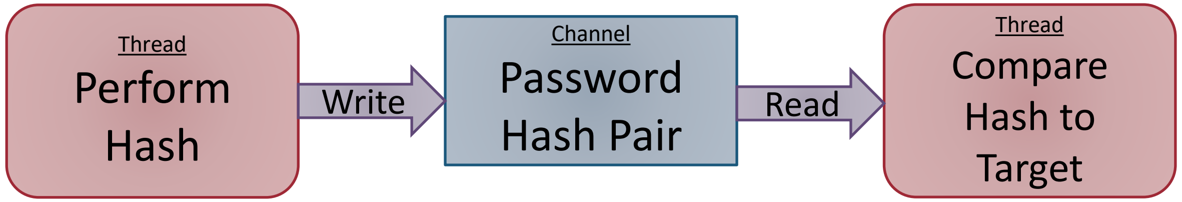
Hash to comparison

Comparison thread tries to read the hash channel

This channel is empty so the thread suspends

Hash channel writes both the hash it has generated and the plaintext used to generate it to the hash channel

This causes the comparison thread to wake up and compare this hash to the provided search hash



Comparison to end search

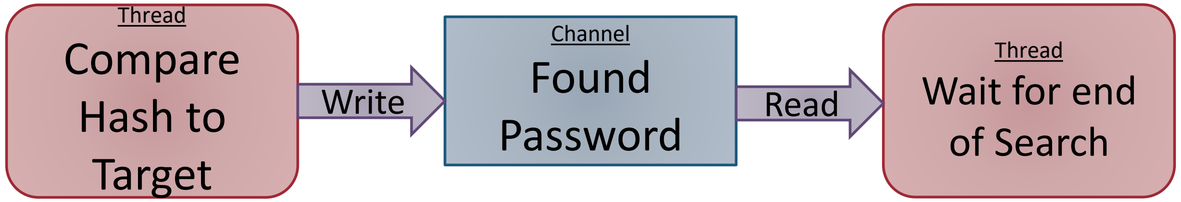
End search thread tries to read the hash channel

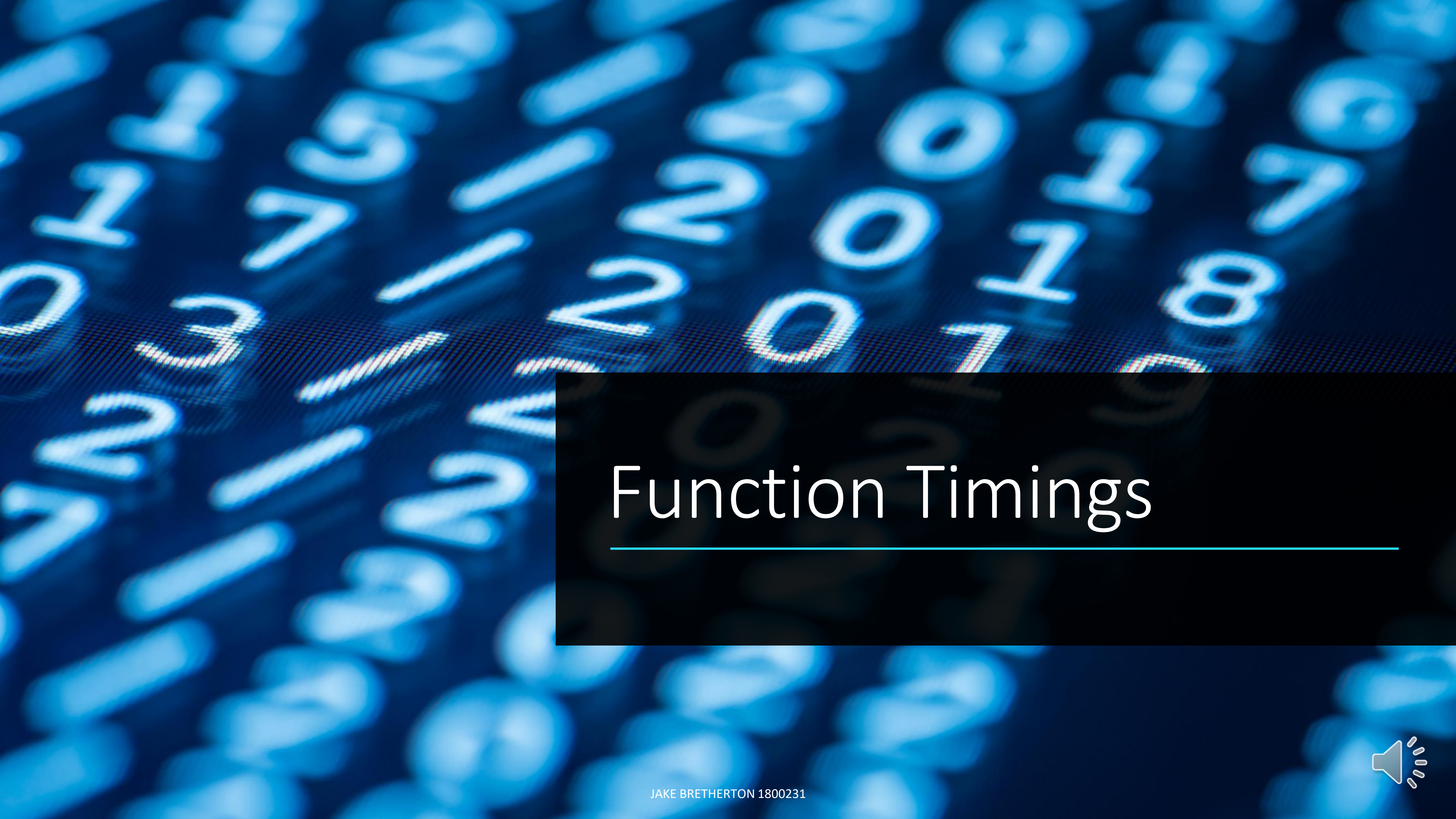
This channel is empty so the thread suspends

If the comparison thread finds the hash did not match the target, it discards the hash and password

If they match then the comparison thread writes the associated password to the end search channel

This causes the end search thread to wake up and stop and join the rest of the threads





Function Timings

JAKE BRETHERTON 1800231

Specs of machine timed on

Processor: Intel Core i5-1035G1, quad core 1GHz

Cache 6MB

Function timer

Wrote a function timer object to run timings on various methods of the program

You can specify the number of timings to perform, and the number of calls to that function per timing

```
// runs timing:           <caller type, return type, argument type(s)>
functionTimer.RunNewTiming<PasswordCracker, string, size_t>(
    "Timing for full password cracking", // name for output file
    &PasswordCracker::CrackPassword, // function to time
    &passwordCracker, // pointer to object function is called on
    iterations, // number of timings
    repititions, // calls per timing
    targetHash // arguments
    );
```

```
// timing for member function
template<class T_caller, class T_ret, class ...T_args>
auto FunctionTimer::RunNewTiming(string name, T_ret(T_caller::*function)(T_args...), T_caller* caller, const int iterations, const int repititions, T_args ...otherArgs)
{
```

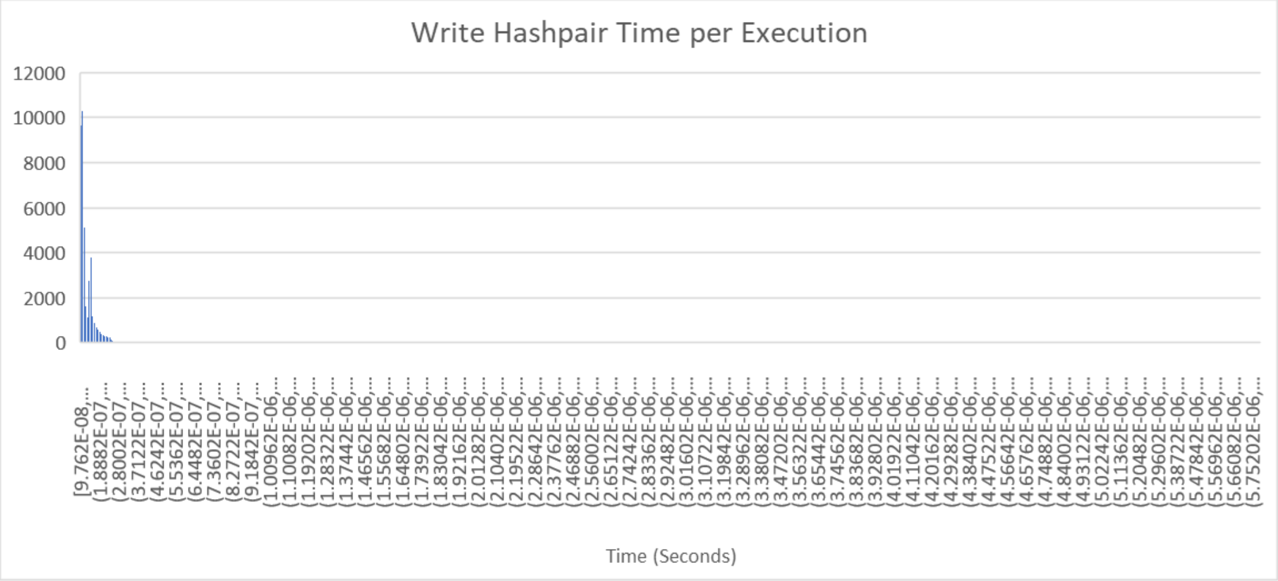
```
inline void CallFunc(T_ret(T_caller::*func)(T_param ...), T_caller * caller,T_param ... params) { (caller->*func)(params ...); }; // calls function pointer
```

Issues with timings

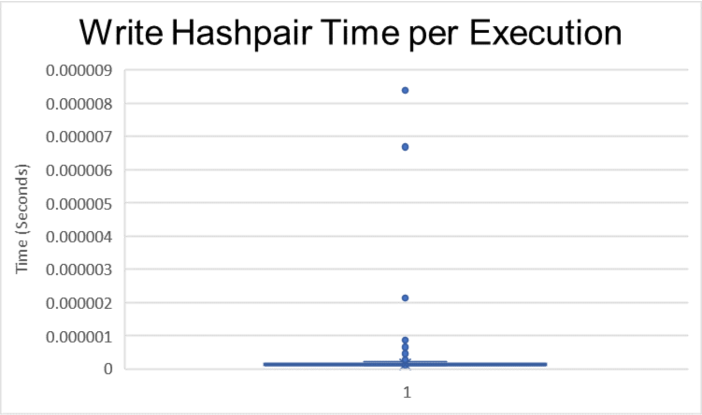
Because the entire solution is fairly complex with many threads that likely spend a significant amount of execution suspended, it was decide to time each task independently. Reading / writing to each channel, and each thread’s work were timed.

Because timing such small segments of code, the high-resolution clock often would not register any time having passed between executions, so many executions were done per timing, and this was repeated many times.

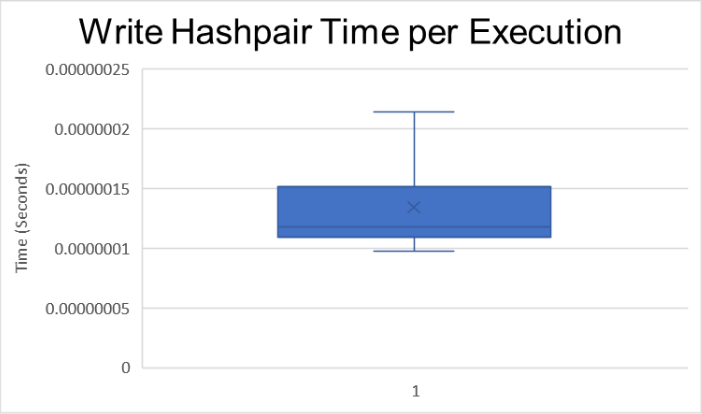
Outliers made histogram graphing difficult, this is a naïve attempt with evenly spaced buckets from the minimum to maximum value



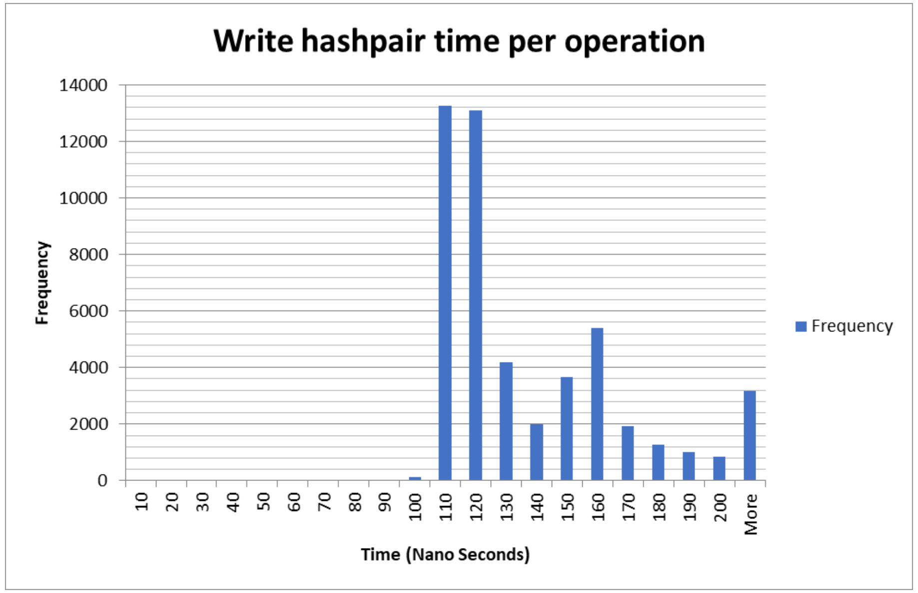
Issues with timings - Solution



The outliers were so extreme they dwarf the rest of the data



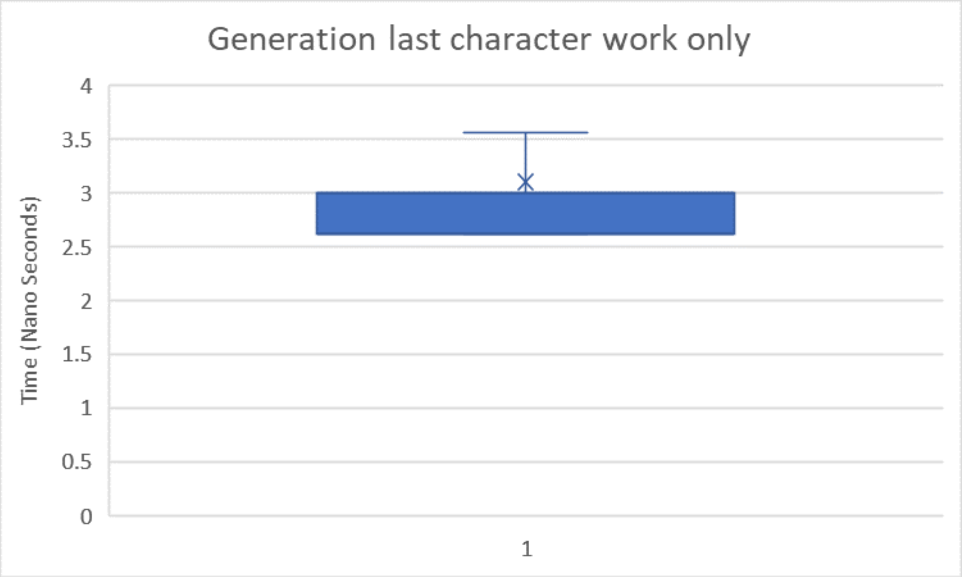
Box and whisker plots were used to visualise the data excluding these outliers.
The Minimum (Q1–1.5*IQR) and Maximum (Q3+1.5*IQR) were then used for the histogram extremes, and outliers all grouped in ‘more’

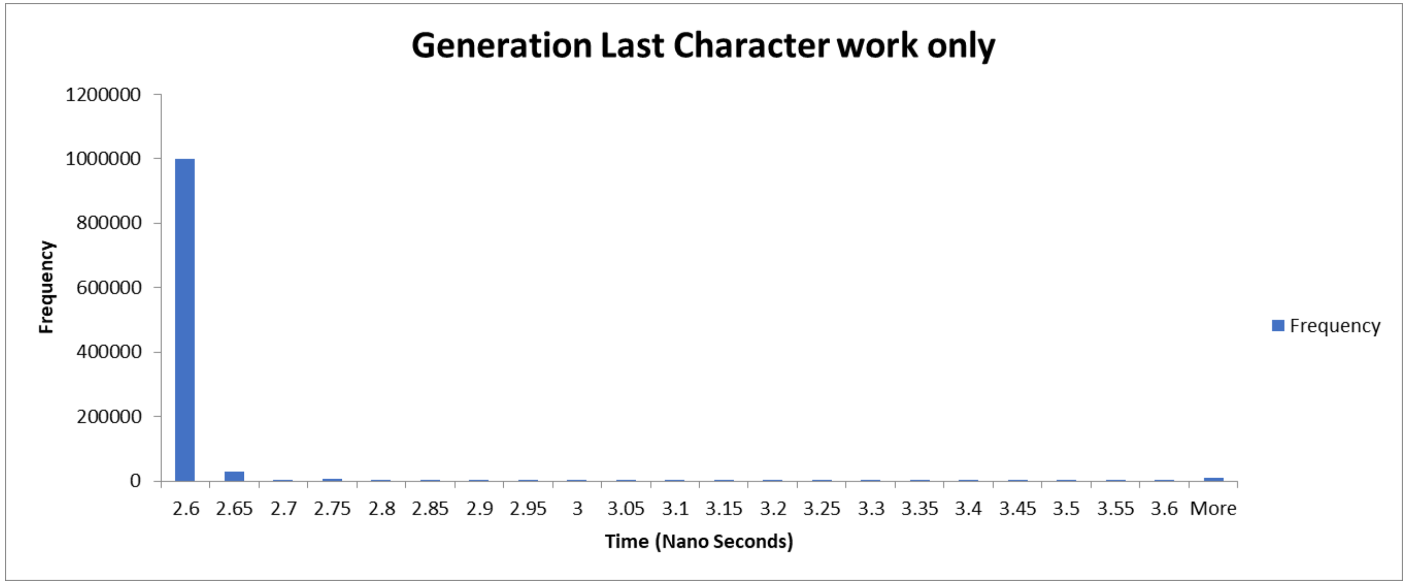


Median Time: 118 Nanoseconds

Generation last character

50,000 timings of 50,000 executions of the last character generation only (no channel writing)
Theoretical time per execution

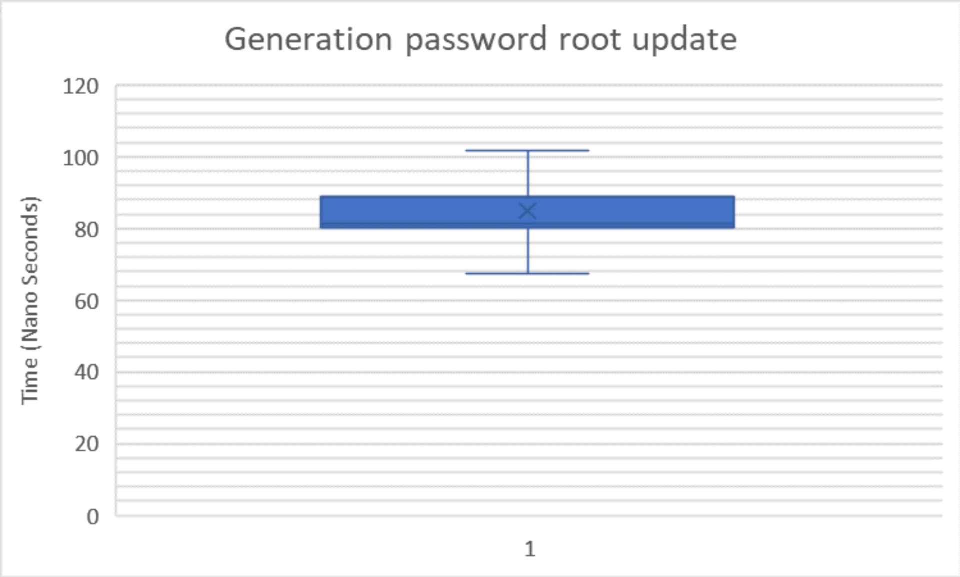


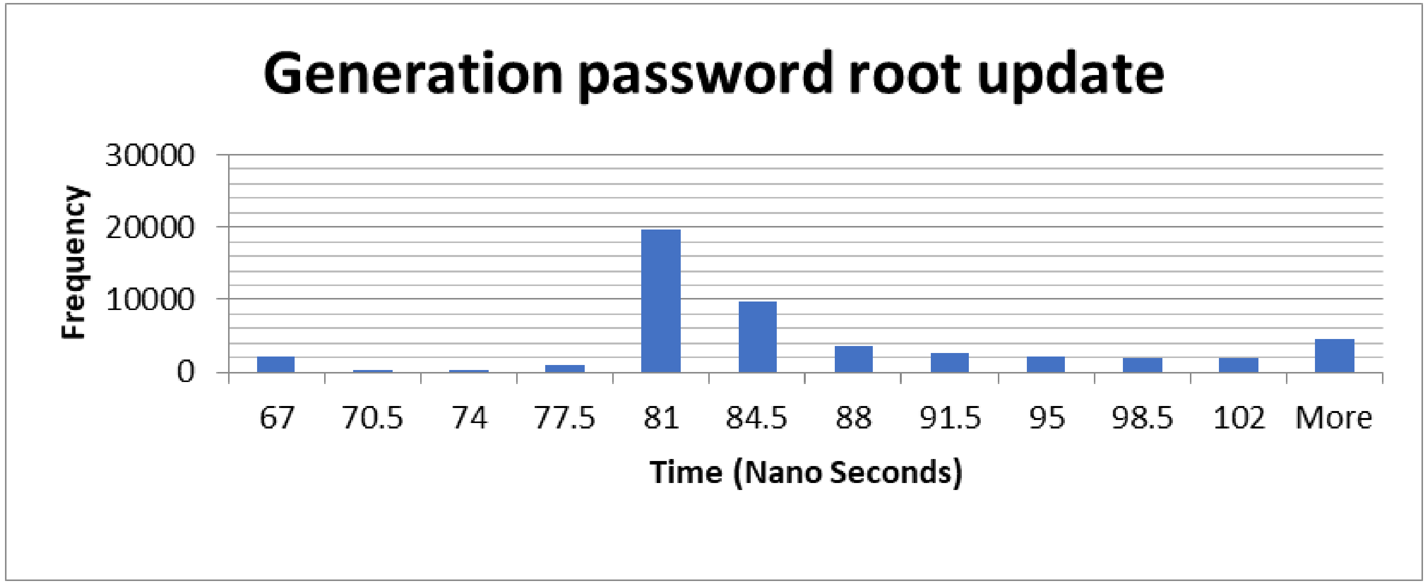


Median Time: 2.62 Nanoseconds

Generation password root

50,000 timings of 50,000 executions of the password root only (no channel writing)
Theoretical time per execution

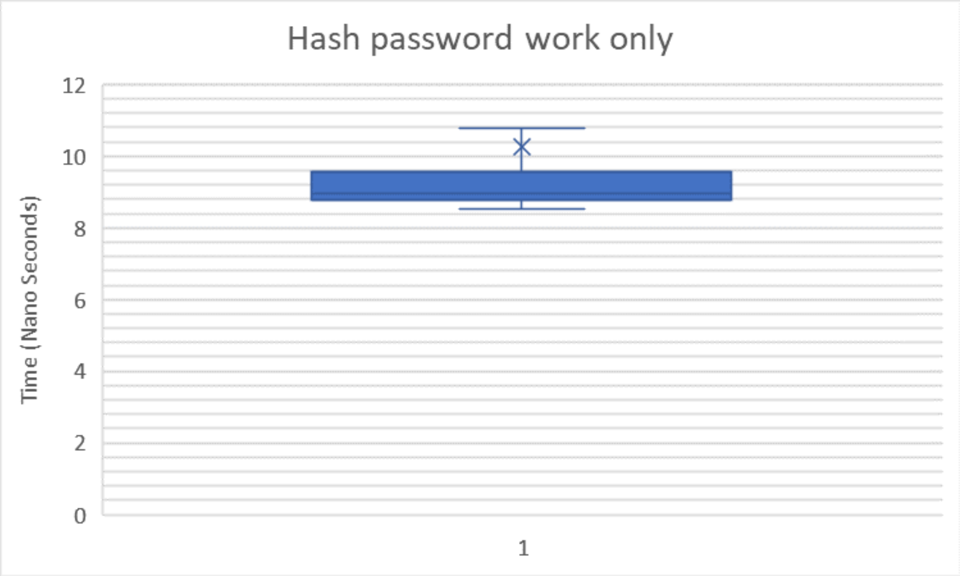


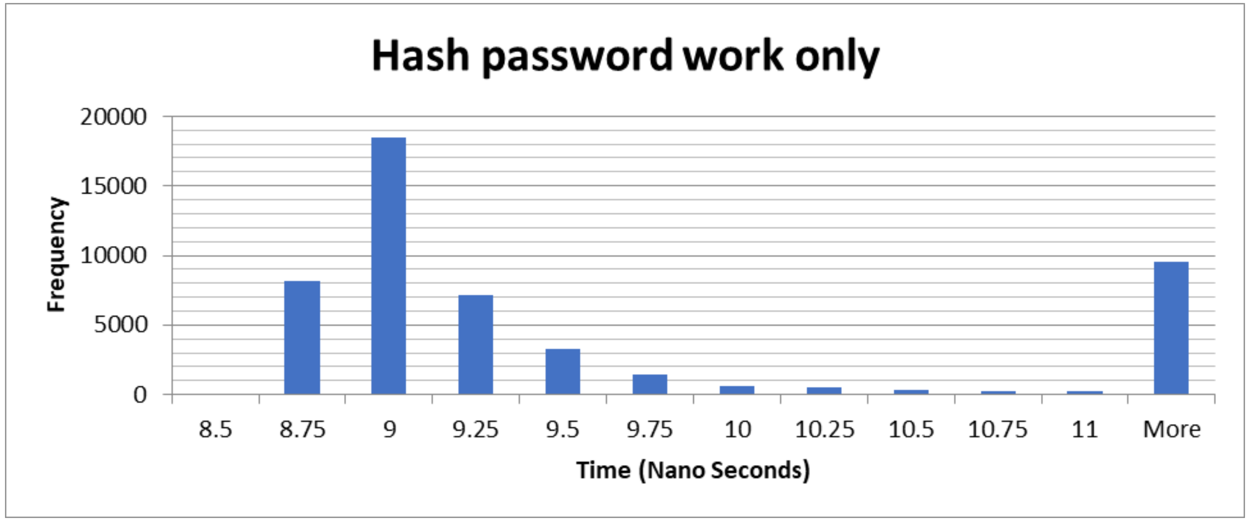


Median Time: 81.3 Nanoseconds

Hash password

50,000 timings of 50,000 executions of password hashing only (no channel read/writing)
Theoretical time per execution

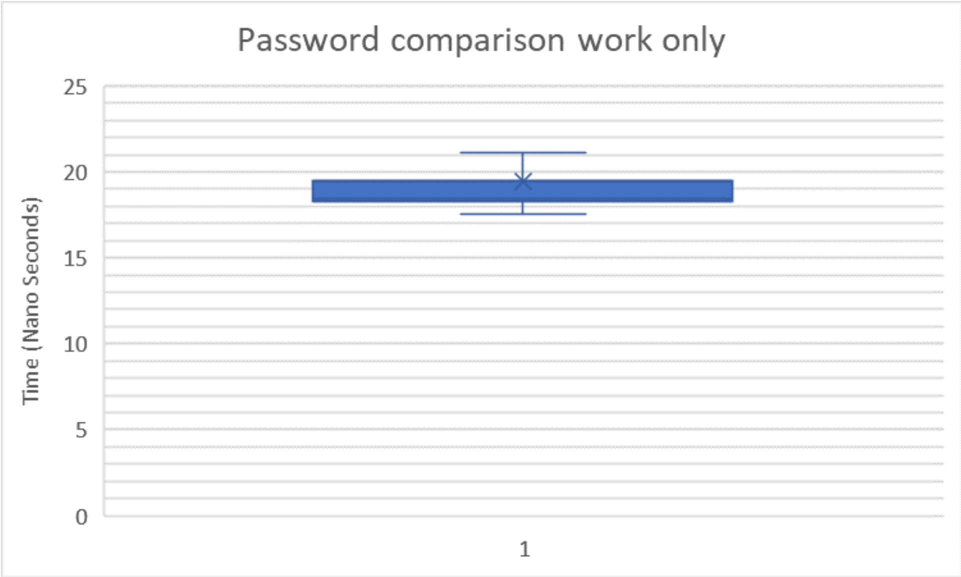


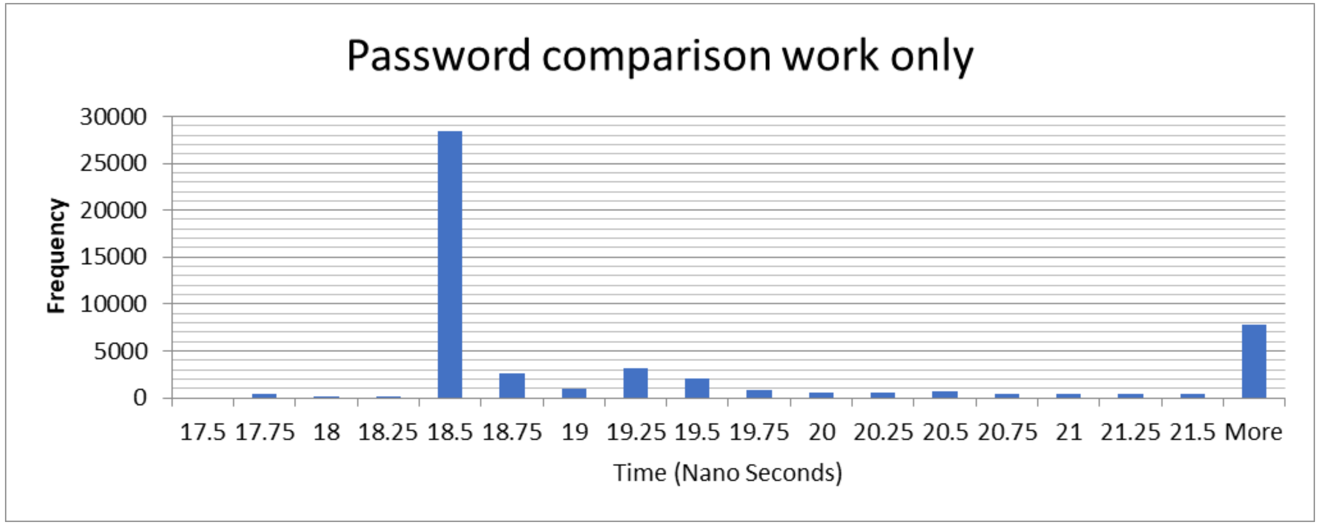


Median Time: 8.94 Nanoseconds

Hash comparison

50,000 timings of 50,000 executions of hash comparison work only (no channel reading)
Theoretical time per execution

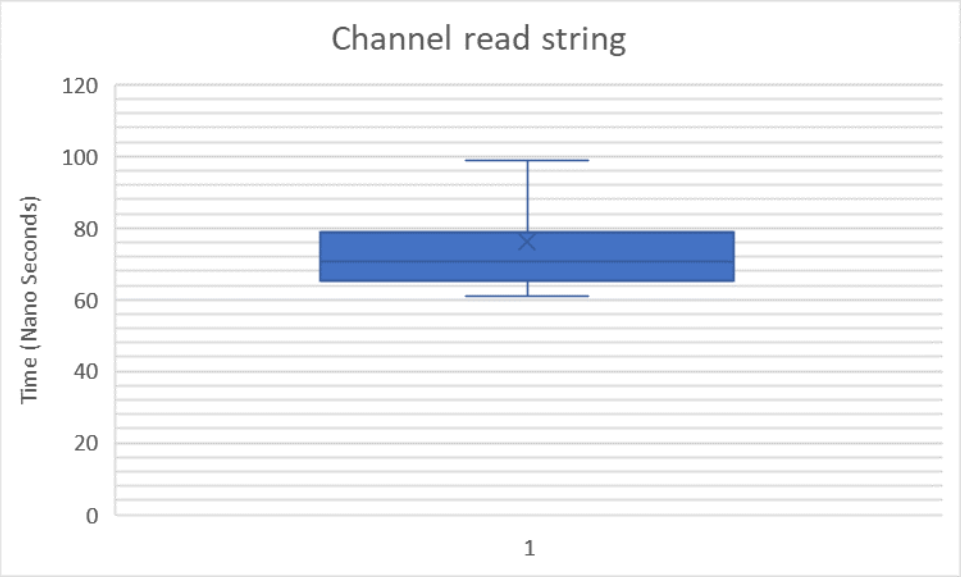


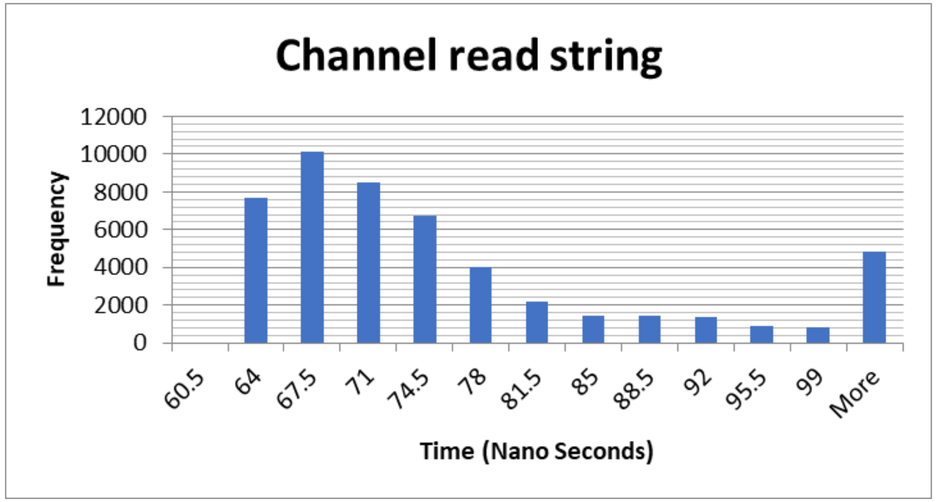


Median Time: 18.44 Nanoseconds

Channel reading string

50,000 timings of 50,000 executions of reading a string from a channel
Theoretical time per execution

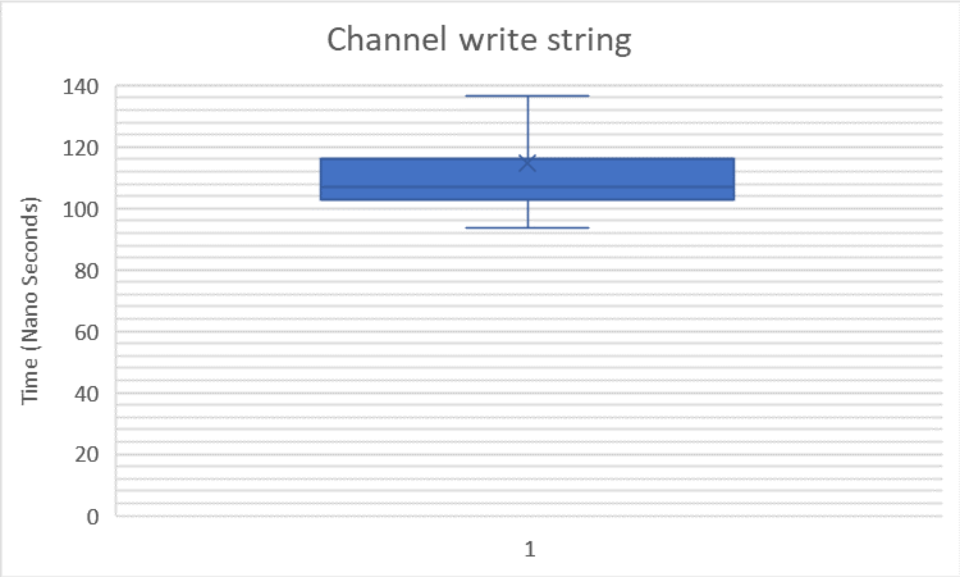


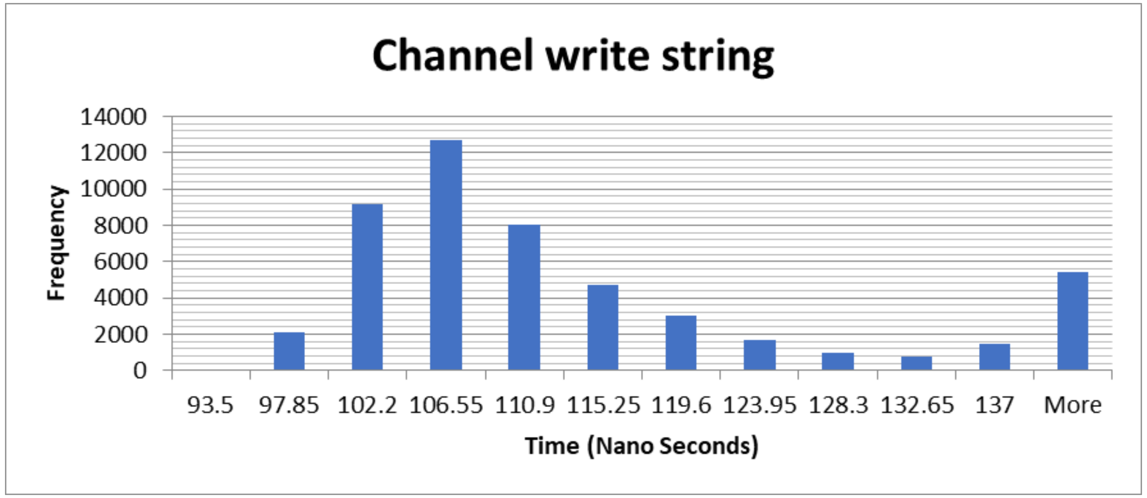


Median Time: 70.56 Nanoseconds

Channel writing string

50,000 timings of 50,000 executions of writing a string to a channel
Theoretical time per execution

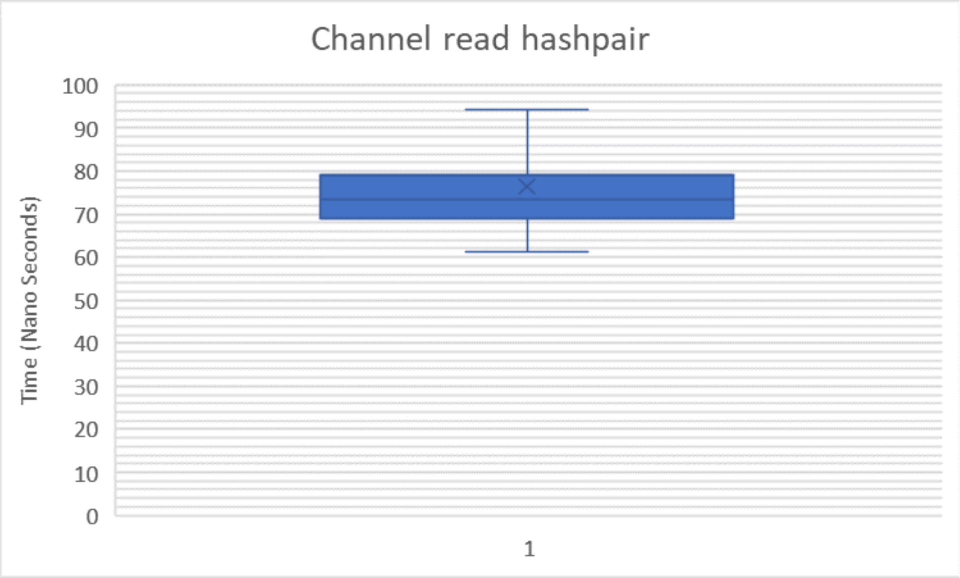


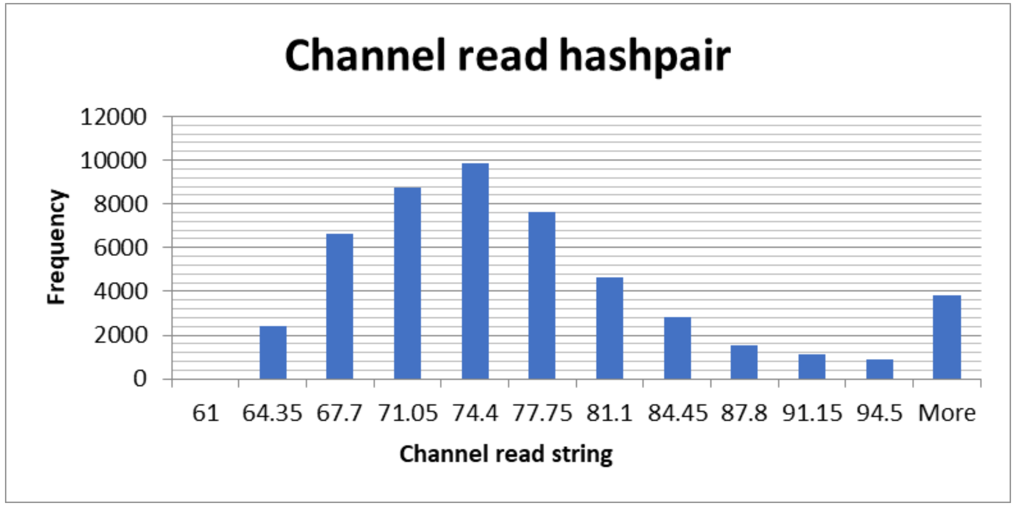


Median Time: 106.98
Nanoseconds

Channel reading hashpair

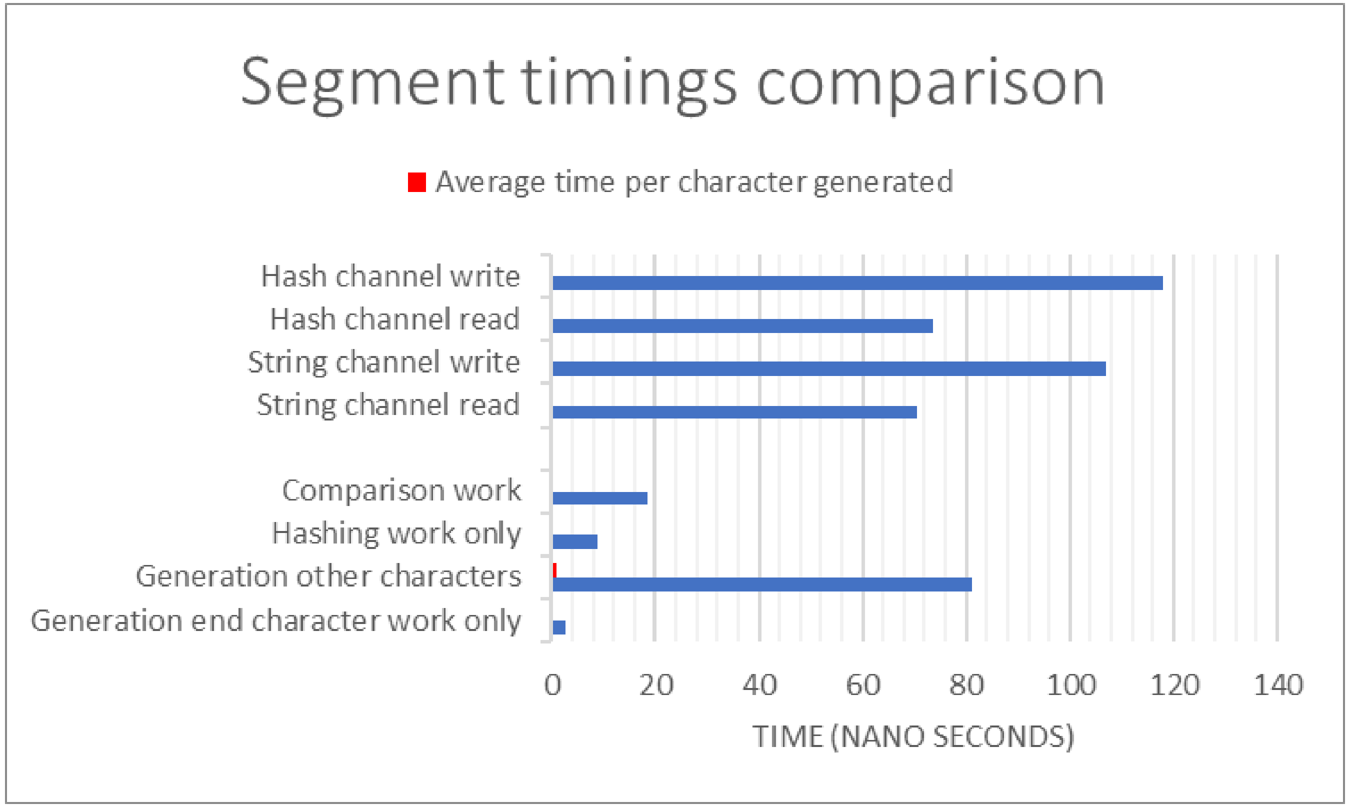
50,000 timings of 50,000 executions of reading a string and hash from a channel
Theoretical time per execution





Median Time: 73.56 Nanoseconds

Comparison

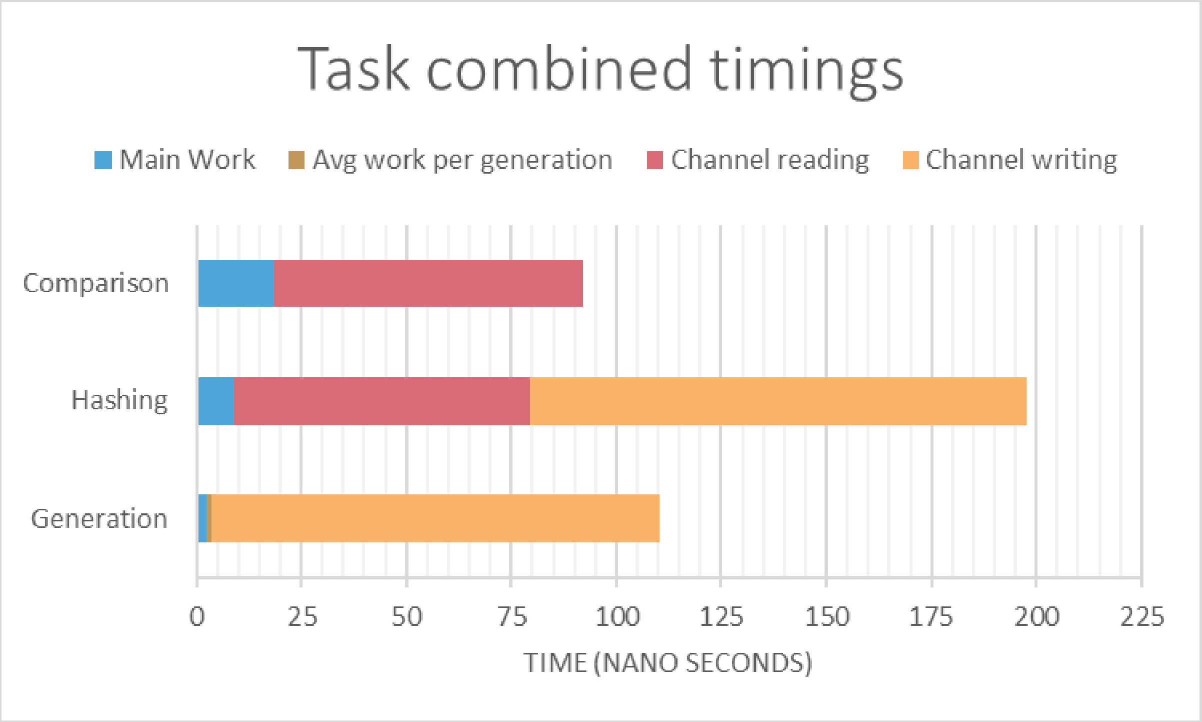


Generation: Single character, string channel writing, 1/95th of root password updating

Hashing: Hash, string channel reading, hashpair channel writing

Comparison: Hash comparison, haspair channel reading

Total comparison



Generation: 110.5ns

Hashing: 197.5ns

Comparison: 92ns

Hashing approximately twice as slow as generation and comparison.

Therefore two threads for ever comparison and generation thread.

Control

Task	Time per guess per thread (Ns)	Threads	Time per guess all threads (Ns)
Password root increment	0.8558	1	0.8558
Password last character increment	220.056	1	220.056
Hashing	395	1	395
Comparison	184	1	184
Total (Theoretical)	799.912	4	799.912

Optimisation

Task	Time per guess per thread (Ns)	Threads	Time per guess all threads (Ns)
Password root increment	0.8558	1	0.8558
Password last character increment	220.056	1	220.056
Hashing	395	2	197.5
Comparison	184	1	184
Total (Theoretical)	799.912	5	602.412

Prediction

Control theoretical time per guess	Optimised theoretical time per guess
799.912	602.412

$$\frac{799.912}{602.412} = \mathit{32.8\%} \text{ Theoretical time improvement}$$

$$Time\ to\ crack\ = ([Theoretical\ time\ per\ guess] * ([number\ of\ characters\ in\ characterset]^{[number\ of\ characters\ in\ password]}))/2$$

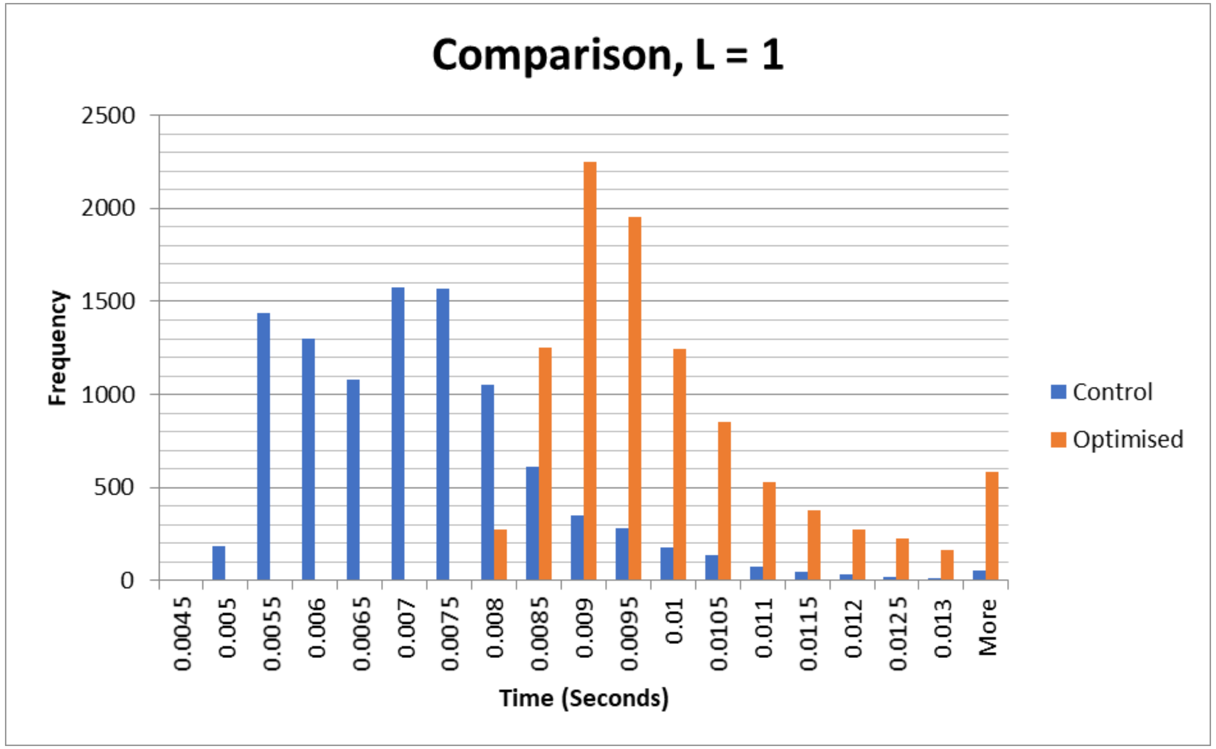
Number of letters in password	Control time predictions	Optimised time predictions
1	38.0μs	28.6μs
2	3.61ms	2.72ms
3	0.343s	0.258s
4	32.6s	24.5s
5	51.6min	38.8min

Results

Number of letters in password	Control time actual	Optimised time actual
1	0.006827s	0.009286s
2	0.011002s	0.012724s
3	0.373407s	0.284701s
4	27.983454s	25.69767s
5	3226.772s	2461.094s

1-3: median of 10,000 timings
4: median of 100 timings
5: median of 3 timings

Results, number of letters = 1



10,000 timings

Timings are fairly erratic

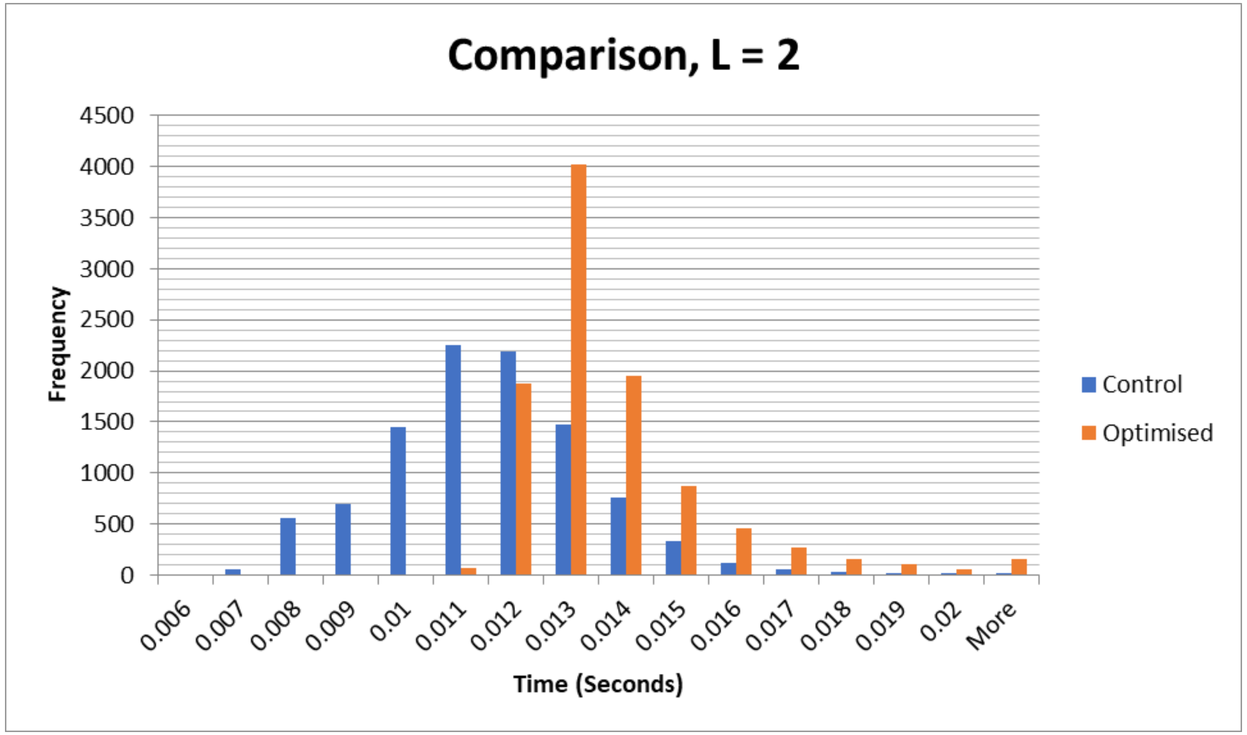
Control has clear advantage over optimisation

Median:

- Control: 0.006827s
- Optimised: 0.009286s

Control 0.002459s (36%) faster

Results, number of letters = 2



10,000 timings

Timings more consistent

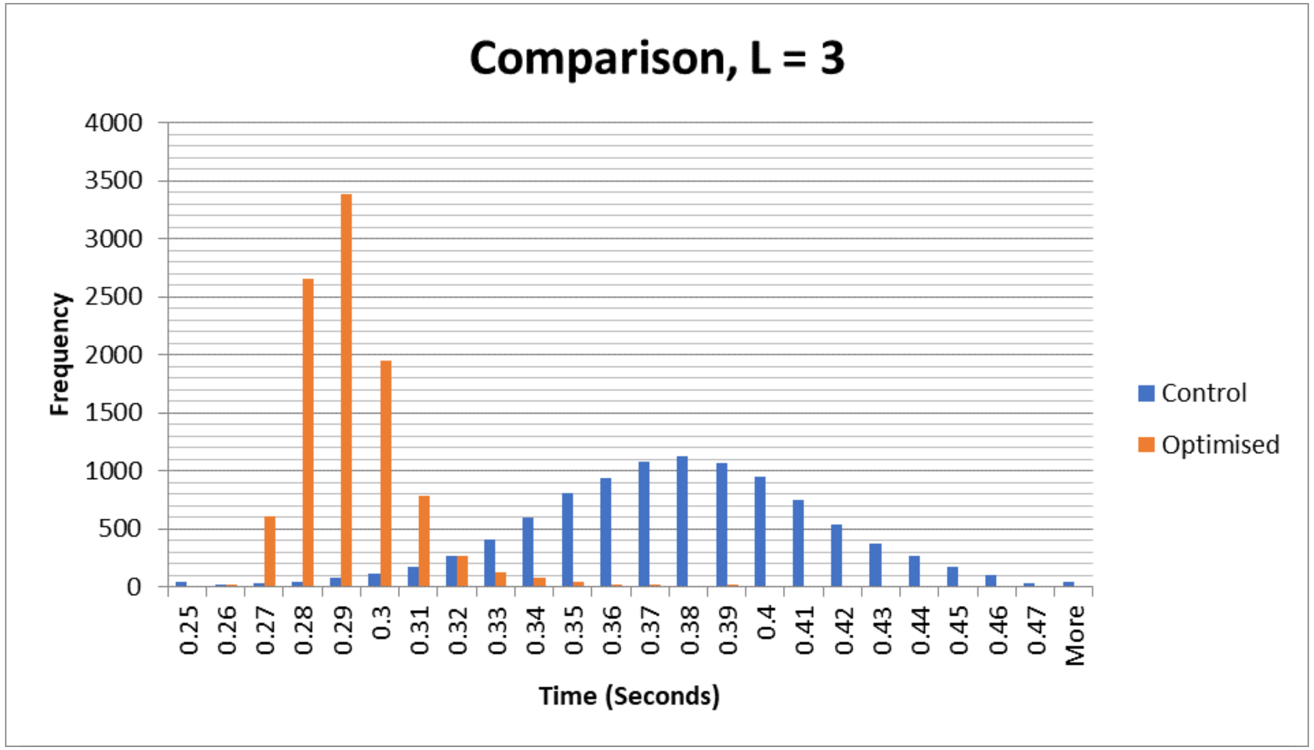
Control still faster than optimisation

Median:

- Control: 0.011002s
- Optimised: 0.012724s

Control 0.001722 s (16%) faster

Results, number of letters = 3



10,000 timings

Timings very consistent

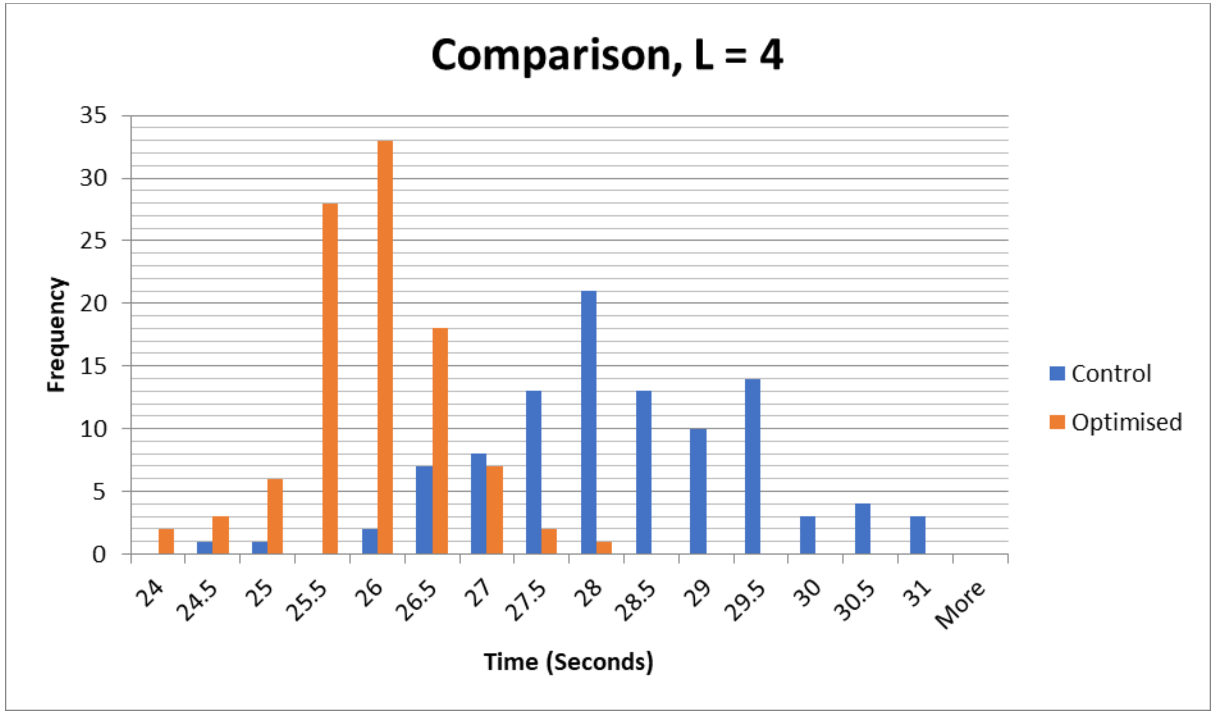
Control still faster than optimisation

Median:

- Control: 0.373407s
- Optimised: 0.284701s

Optimisation 0.08871s (31%) faster, in line with predictions

Results, number of letters = 4



100 timings

Far less statistically reliable but necessary due to time constrains

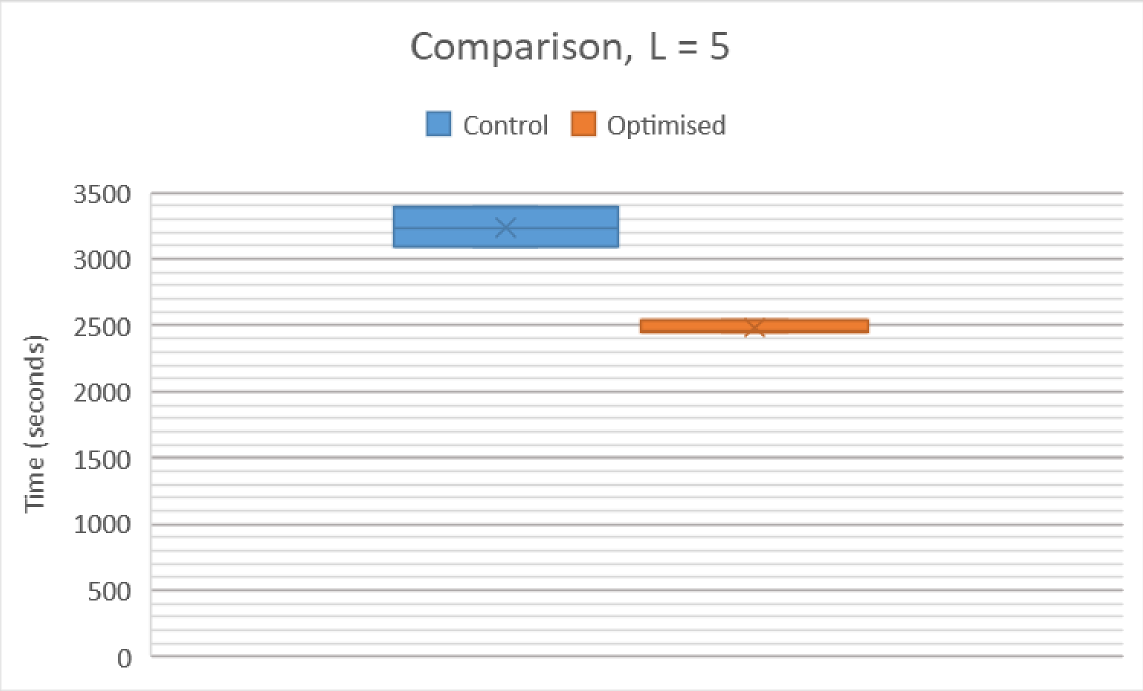
Still fairly statistically rigorous

Optimisation clearly has advantage

Median:
- Control: 27.98345s
- Optimised: 25.69767s

Optimisation 2.285784s (9%) faster

Results, number of letters = 5



3 timings

Not statistically rigorous but useful as indication

Optimisation clearly faster

Median:

- Control: 3226.772s (00:53:47)
- Optimised: 2461.094s (00:41:01)

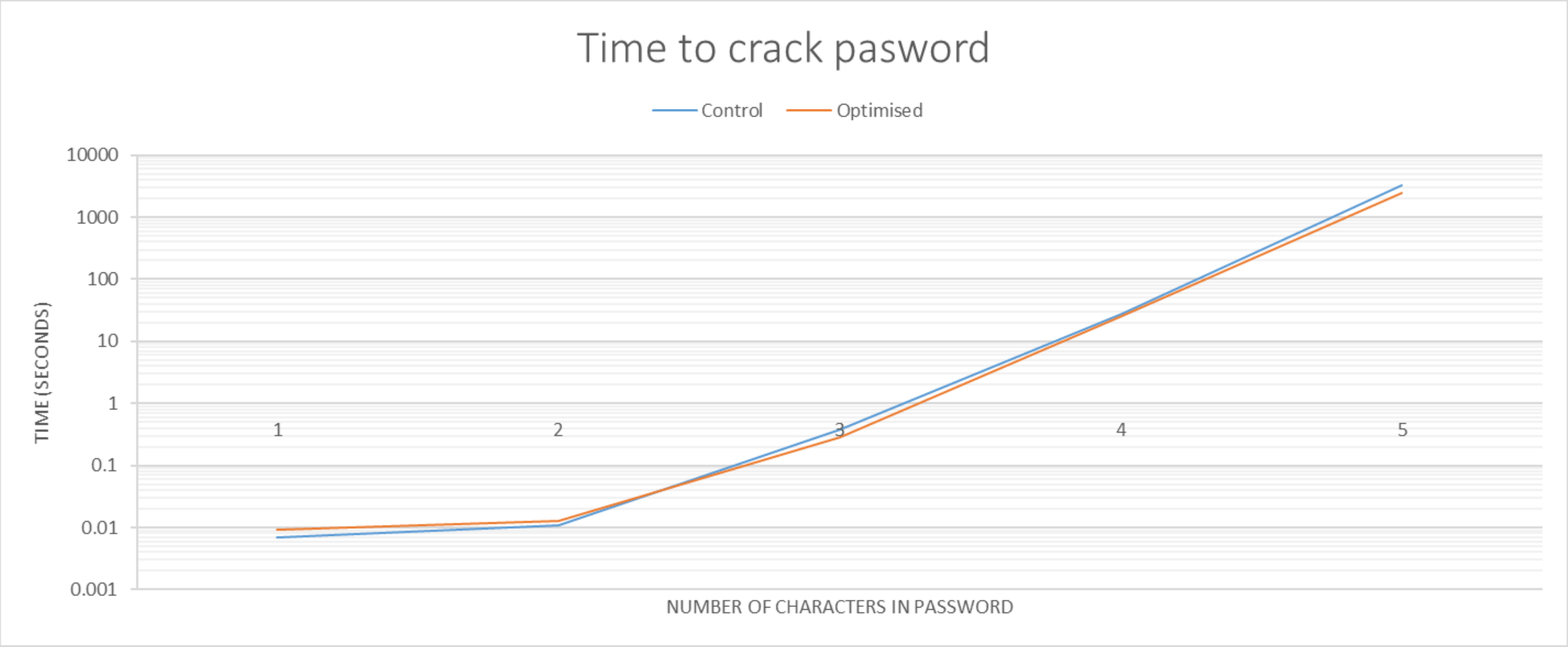
Optimisation 765.6775s (00:12:46) (31%) faster, in line with predictions

Difference from prediction

Length of password	Predicted (seconds)	Actual (seconds)	Percentage error
Control: L =1	3.79958E-05	0.006827	17867.78%
L = 2	0.003609601	0.011002	204.80%
L = 3	0.342912095	0.373407	8.89%
L = 4	32.57664903	27.983454	-14.10%
L = 5	3094.781657	3226.771973	4.26%
Optimised: L = 1	2.86146E-05	0.009286	32352.02%
L = 2	0.002718382	0.012724	368.07%
L = 3	0.258246314	0.284701	10.24%
L = 4	24.53339981	25.69767	4.75%
L = 5	2330.672982	2461.094482	5.60%

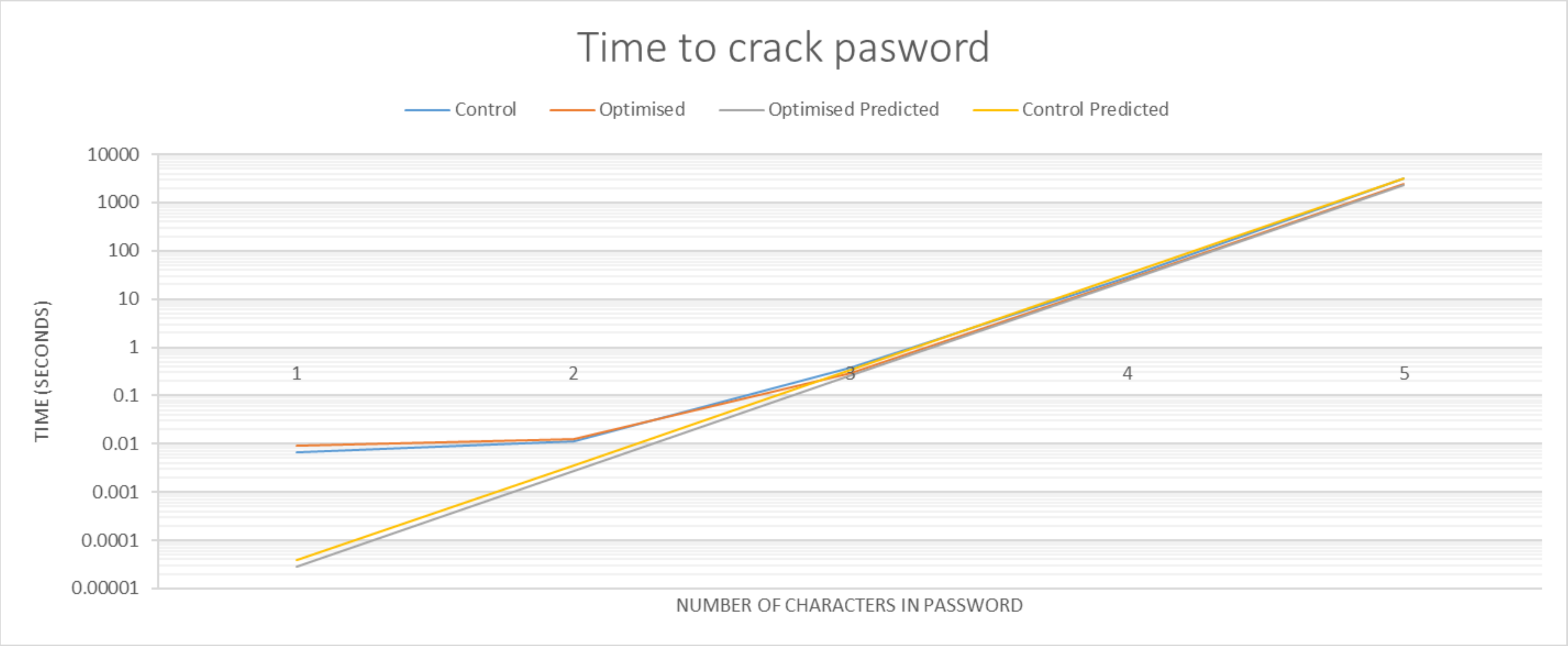
Results

For number of characters 3-5 the optimisation was effective and timings were within 5-15% of what was predicted. However, for lower numbers the program did not behave at all as theorised and the optimisations made it run slower.



Results

When compared to the predicted it is clear there is some floor of how fast the solution can operate, at which it deviates from the theoretical time.



Explanation

This is likely due to the unpredictable yet significant amount of time it takes to launch a thread, this would also explain as to why the optimisation performed *worse* than the control for lower lengths; it had an additional thread to launch.

The allocation of memory and construction of channels and barriers would have a constant time cost that would become an increasingly small proportion of the execution time as the length of characters increase but would be a significant amount of execution time for the far faster running calls of one or two character password guesses.

At the low times a matter of milliseconds it could take of overhead for the operating system to get around to launching a new thread would have significant effects on the timings.

Conclusion

I would therefore say the optimisations with the thread arrangement offers a significant advantage, approaching a 32% improvement, over the naïvely segmented approach for non-trivial lengths of password.

Questions

1800231@uad.ac.uk