

# CS 1699: Privacy in the Electronic Society

## *Project 1 – Side-Channel Attacks*

Jonathan Dyer

February 26, 2018

### Task W0

#### Problem P: String Comparison

The problem chosen for this assignment is *string comparison*, which involves checking one string for equality with another string. This includes comparing both the length and the content of each string for a mismatch.

**Input:** Two strings to be compared.

**Output:** Boolean indicating if the strings are *equal*.

### Task W1

String comparison is a seemingly innocuous problem, but has big implications for privacy. Comparing values is a basic, necessary task for any system involving authentication or user input. An application may check if a given password is correct by comparing its string value to the stored password for a user, whether in plaintext or by comparing their hash values. This can occur via a custom comparison for the object being compared or by using simple string comparison (both have been done historically). For example, the Java `MessageDigest.isEqual` method uses a simple byte comparison to check whether two digests are equal, which is essentially the implementation of string comparison for some languages.

### Task W4

#### Algorithm A: Break-on-Inequality

Firstly, let's examine **Algorithm A** so that we can understand how and why this naive algorithm varies in runtime (including for inputs of the same size). In the pseudocode below, first notice that if the strings are different lengths, a 'False' is returned immediately (lines 5-6). This clearly changes the runtime in the case of different-sized inputs, which can reveal private information as discussed below. More subtle is the character-by-character comparison, which breaks as soon as it finds a mismatch between the two strings (line 11). This reveals info about *how* close the two strings are (or in the case of an attacker, how correct the guess was).

```
1 // This method takes two strings, a and b, and returns True if they are equal, False otherwise
2 def algorithm_A (String a, String b)
3
4     // First check that the lengths are the same
5     if a.length != b.length
6         return False
7
8     // Now iterate through characters, comparing one by one
9     int i = 0
10    while i < a.length
11        if a[i] != b[i]
12            return False
13        i = i+1
14
15    // If we make it all the way, they match!
16    return True
```

## Results

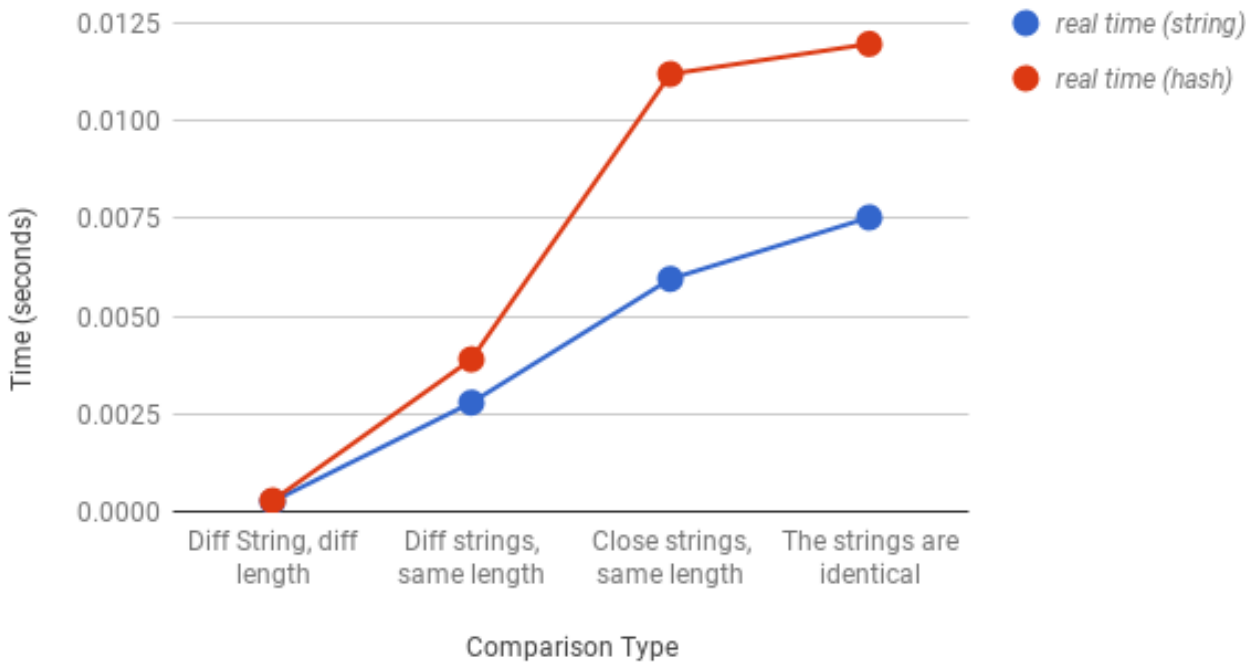
Benchmarking the above code—implemented in Ruby—over 1000s of iterations gives the results displayed in the table below. Note that the "different strings of the same length" differed in the first or nearly first characters while the "close strings of the same length" differed only on the final character, for both the sentence and the hash value. The following observations (about real time taken to compare inputs) are plain from the data:

- Comparing different strings of the same length is an order of magnitude slower than strings of different length.
- Comparing nearly-identical strings is *another* order of magnitude slower.
- The difference between two close strings vs. two of the same string is very small, although becomes more noticeable with a longer input.

|       |                                  |                    |                    |                     |         |
|-------|----------------------------------|--------------------|--------------------|---------------------|---------|
| Naive | This is a good dog               |                    |                    |                     |         |
|       | Test with string                 | real time (string) | iterations in 5.0s | iterations / second | slower? |
|       | Diff String, diff length         | 0.000255           | 16.386M            | 3.263M              |         |
|       | Diff strings, same length        | 0.002788           | 1.936M             | 386.523k            | 8.44x   |
|       | Close strings, same length       | 0.005953           | 860.031k           | 170.027k            | 19.19x  |
|       | 286755fad04869ca523320acce0dc6a4 |                    |                    |                     |         |
|       | Test with hash                   | real time (hash)   | iterations in 5.0s | i / s               | slower? |
|       | Diff String, diff length         | 0.000282           | 16.203M            | 3.207M              |         |
|       | Diff strings, same length        | 0.003896           | 1.423M             | 280.709k            | 11.43x  |
|       | Close strings, same length       | 0.011195           | 534.558k           | 104.985k            | 30.55x  |
|       | The strings are identical        | 0.011967           | 518.313k           | 103.680k            | 30.94x  |

This may also be clearer from the following graph of the real-time taken by each comparison type. Notice that the difference between two different strings of the same length and two close strings is more dramatic as the length increases.

### Naive Comparison



## Effect on Privacy

It is clear that this measurable difference in string comparison times can be leveraged in a timing attack against some systems that make use of such an algorithm. Specifically, if there is ever a user-facing application that requires input that must be verified, such as a username or password, some type of comparison will be involved. Consider such a system, with a typical login page where a user might enter their credentials, which will then be compared to the system's stored list of credentials using the above naive string comparison. A *chosen plaintext* attack is possible against this portal, say to discover valid usernames. The attacker need only perform these steps:

1. For any given set  $x$  of known characters and  $y = y_0 \dots y_n$  unknown, try every possibility for a variable  $a$  in the input  $x||a||y_1 \dots y_n$  and some fixed filler, say `<underscore>`, for the rest of  $y$ .
2. Sample some dozens (or hundreds) of times on each combination, and then take the mean (or median to mitigate outliers) of the time taken to return an error/False message.
3. Set  $a$  to be the character/byte with the greatest mean/median value, add it to  $x$ , and then return to step 1.

By repeating this process, first to find the correct length and then to find the correct combination of characters, an attacker could potentially:

1. Discover valid usernames for the system.
2. Recover full login info (including passwords) for users.

This may require a more sophisticated timing software than Ruby's built-in benchmarking module, but I believe that it would not be difficult to achieve such an attack, even with (for example) a basic timing program.

## Task W7

### Algorithm B: Constant-Time Comparison

Now let's examine **Algorithm B** so that we can understand how and why this revised algorithm runs in a more constant-time manner, mitigating some of the effects discussed for Algorithm A above. In the pseudocode below, first notice that before anything else we select a consistent number of iterations to run the byte-wise comparisons for (`len` in the code below, line 5)—this helps ensure that we don't reveal information about the length of the secret value (perhaps a login name), since that was an obvious and large timing difference in our naive algorithm. Next, consider the revised loop through our inputs (lines 9-13). Rather than breaking this loop once we find a mismatch, we continue for the specified length `len` (line 9). Then we check that both of the next characters are even valid to be compared (line 10). If so, we skip to the `else` (line 12) and proceed with the comparison. If either one is *not* valid, we execute a "dummy" comparison to (hopefully) take the same amount of time as a real one (line 11). Finally, we return the resulting boolean.

```
1 // This method takes two strings, a and b, and returns True if they are equal, False otherwise
2 def algorithm_B (String a, String b)
3
4   // We'll assume string a is the "system string" we're comparing the user-input against
5   int len = a.length
6
7   // Now set a boolean and then iterate through characters, comparing each (even on mismatch)
8   boolean bool = (a.length == b.length)
9   for i in 0... len // for every character in the fixed length
10    if a[i] == null || b[i] == null // if we've reached the end of one string
11      bool = (a[0] == a[0]) && bool // compare something anyways
12    else // otherwise
13      bool = (a[i] == b[i]) && bool // compare the two chars, update the boolean
14
15   return bool
```

## Results

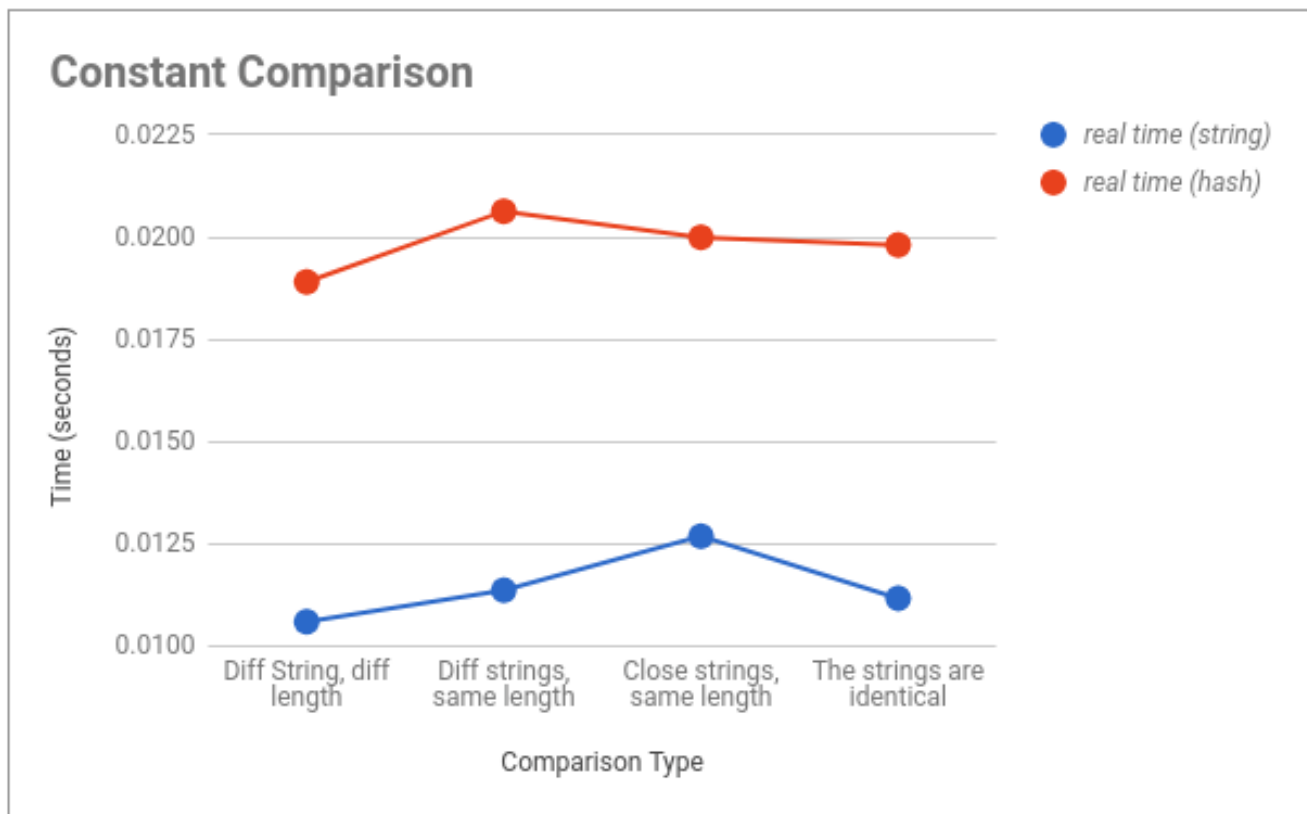
Benchmarking the above code—implemented in Ruby—over 1000s of iterations gives the results displayed in the table below. Note that the timing distinctions between the different types of comparison are almost entirely eliminated for shorter strings, and are statistically insignificant for longer strings/hash values. It is worth noting that comparing strings of different length still seems to be notably faster, especially for shorter inputs. This could reveal the length of a secret, given enough samples.

| Constant                   | This is a good dog |                    |         |         |  |
|----------------------------|--------------------|--------------------|---------|---------|--|
| Test with string           | real time (string) | iterations in 5.0s | i / s   | slower? |  |
| Diff String, diff length   | 0.010596           | 495.144k           | 97.921k |         |  |
| Diff strings, same length  | 0.011366           | 431.600k           | 85.590k | 1.14x   |  |
| Close strings, same length | 0.012686           | 431.400k           | 86.376k | 1.13x   |  |
| The strings are identical  | 0.011168           | 439.416k           | 87.895k | 1.11x   |  |

|                            | 286755fad04869ca523320acce0dc6a4 |                    |         |              |  |
|----------------------------|----------------------------------|--------------------|---------|--------------|--|
| Test with hash             | real time (hash)                 | iterations in 5.0s | i / s   | slower?      |  |
| Diff String, diff length   | 0.018902                         | 259.632k           | 52.482k |              |  |
| Diff strings, same length  | 0.020629                         | 255.500k           | 51.108k | within error |  |
| Close strings, same length | 0.019993                         | 260.355k           | 51.922k | within error |  |
| The strings are identical  | 0.019806                         | 263.648k           | 52.152k | within error |  |

The improvement is especially clear given the graph below, which shows much more even timing results across different types of input. In particular, the test on longer (hash) strings shows a slight reversal in the trend across strings of the same length—which could indicate that it would lead an attacker to make incorrect guesses (a desirable trait).



## Trade-Offs & Design Decisions

There is a significant performance cost associated with making constant-time comparisons, especially when trying to obscure even the length of the secret value(s). Notice in the graphs above (INCLUDE EXTRA GRAPH BELOW THAT COMPARES THE TWO) for Naive vs. Constant Comparison that the constant-time algorithm takes nearly twice as long when comparing similar strings and *several times* as long when comparing very different strings. This would be a highly significant cost on a system or web-server that may receive thousands of requests a day/hour/minute.

Additionally, not all the timing difference for comparing strings of different length is mitigated. This could be because the 'else' conditional code (lines 12-13) is not being executed for every null character of the compared

string, resulting in faster overall execution. Some amount of randomness could perhaps be introduced that would further obscure those lines of comparison.

Finally, because the chosen amount of time is based on the *first* string passed to the algorithm, we risk revealing length of secret values anyways if our implementation is known and the attacker can duplicate our system/working environment.

## **Task W8**

### **Remaining Problems**

There is still potential to leak lengths of secret values, and there may be a better way to obscure those lengths that I have not implemented above. -j THIS WILL BE EXPANDED