Scalable Comparison of JavaScript V8 Bytecode Traces

Javier Cabrera Arteaga
KTH Royal Institute of Technology
Stockholm, Sweden
javierca@kth.se

Martin Monperrus
KTH Royal Institute of Technology
Stockholm, Sweden
martin.monperrus@csc.kth.se

Benoit Baudry
KTH Royal Institute of Technology
Stockholm, Sweden
baudry@kth.se

Abstract

The comparison and alignment of runtime traces are essential, e.g., for semantic analysis or debugging. However, naive sequence alignment algorithms cannot address the needs of the modern web: (i) the bytecode generation process of V8 is not deterministic; (ii) bytecode traces are large.

We present STRAC, a scalable and extensible tool tailored to compare bytecode traces generated by the V8 JavaScript engine. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk. STRAC can identify semantically equivalent web pages and is capable of processing huge V8 bytecode traces whose order of magnitude matches today's web like https://2019.splashcon.org, which generates approx. 150k of V8 bytecode instructions.

CCS Concepts • **Information systems → World Wide Web**; • **Theory of computation → Program semantics**; • **Software and its engineering →** *Interpreters*; *Source code generation*; *Designing software.*

Keywords V8, Sequence alignment, JavaScript, Bytecode, Similarity measurement

ACM Reference Format:
Javier Cabrera Arteaga, Martin Monperrus, and Benoit Baudry. 2019. Scalable Comparison of JavaScript V8 Bytecode Traces. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '19), October 22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358504.3361228

VMIL '19, October 22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6987-9/19/10…$15.00
https://doi.org/10.1145/3358504.3361228

1 Introduction

Runtime traces record the execution of programs. This information captures the dynamics of programs and can be used to determine semantic similarity [29], to detect abnormal program behavior [8], to check refactoring correctness [22] or to infer execution models [1]. In many cases, this is achieved by comparing execution traces, e.g. comparing the traces of the original program and the refactored one. The comparison of program traces can be based on information retrieval [17], tree differencing [9, 27] and sequence alignment [2, 11]. In this paper, we focus on the latter, in order to compare sequences of V8 bytecode instructions resulting from the execution of JavaScript code.

V8 is an open source, high-performance JavaScript engine. For debugging purposes, it provides powerful facilities to export page execution information [21], including intermediate internal bytecode called the V8 bytecode [4].

Due to the dynamic nature of the Web, we observe that the bytecode generation process of V8 is not deterministic. For example, visiting the same page several times results in different V8 bytecode traces every time. This nondeterminism is a key challenge for sequence alignment approaches, even if they perform well on deterministic program traces [10]. Besides, V8 bytecode traces are large. Naive sequence alignment algorithms are time and space quadratic on trace sizes and do not scale to V8 bytecode traces. To illustrate this scaling problem, let us consider a simple query to https://2019.splashcon.org: it generates between 139555 and 162558 V8 bytecode instructions, and aligning two traces of such size, requires approximately 150GB of memory[1]. This memory requirement is not realistic for trace analysis tasks on developer's personal computers or servers. The key challenge that we address in this work is to provide a trace comparison tool that scales to V8 bytecode traces.

In this paper, we present STRAC (Scalable Trace Comparison), a scalable and extensible tool tailored to compare bytecode traces from the V8 JavaScript engine. STRAC implements an optimized version of the DTW algorithm [18]. Given two V8 bytecode traces and a distance function between trace events, STRAC computes and provides the best alignment. The key insight is to split access between memory and disk.

Our experiments compare STRAC with 6 other publicly-available implementations of DTW. The comparison involves

[1] In this paper, memory means RAM.

100 pairs of V8 bytecode traces collected over 6 websites. Our experimental results show that 1) STRAC can identify semantically equivalent web pages and 2) STRAC is capable of processing big V8 bytecode traces whose order of magnitude matches today's web.

To sum up, our contributions are:

- An analysis of the challenges for analyzing browser traces, due to the JavaScript engine internals and the randomness of the environment. We explain and show examples of how the same browser query can generate two different V8 bytecode traces.
- A tool called STRAC that implements the popular alignment algorithm DTW in a scalable way, publicly available at https://github.com/KTH/STRAC .
- A set of experiments comparing 100 V8 bytecode traces collected over 6 real world websites:google.com, kth.se, github.com, wikipedia.org, 2019.splashcon.org and youtube.com. Our experiments show that STRAC copes with the non-deterministic traces and is significantly faster than state-of-the-art tools.

The paper is structured as follows. First we introduce a background of V8 bytecode generation non-determinism and the formalisms used in our work (Section 2). Then follows with technical insights to implement STRAC (Section 3), research question formulation, experimental results with a discussion about them (Section 4). We then present related work (Section 5) and conclude (Section 6).

2 Background

In this section we discuss the key insights behind the non-determinism of the V8 bytecode generation process, as well as the foundations of the DTW alignment algorithm.

2.1 Browser Traces

Our dynamic analysis technique is evaluated with V8 bytecode [19]. In this subsection, we describe how the V8 engine generates bytecode trace. We collect such traces to evaluate our trace comparison tool. In this work, we use the term "V8 bytecode trace" to refer to the result of executing V8 with the *–print-bytecode* flag [21].

2.1.1 V8 Bytecode Generation

The V8 engine compiles JavaScript source code to an intermediate representation called "V8 bytecode". This is done to increase execution performance. The V8 engine parses and compiles every JavaScript code declaration present in HTML pages into a bytecode representation, composed by function declarations, like the one shown in Figure 1. These function declarations came from V8 builtin JavaScript code and external JavaScripts.

V8's bytecode interpreter is a register machine [16]. Figure 1 shows a JavaScript code and its bytecode translation. Each bytecode operator specifies its inputs and outputs as register operands. V8 has 180 different bytecode operators.

The bytecode translation is lazy, i.e. V8 tries to avoid generating code it "thinks" might not be executed. Consequently, a function that is not called will not be compiled [28]. For example, removing line 2 in the top listing of Figure 1 would prevent the compilation of bytecode for the function declared in line 1. This behavior has an impact on the collected traces.

```
1  function plusOne(a){ return a.value + 1; }
2  plusOne( {value : 2018} );
```

```
1  [generated bytecode for function: plusOne]
2  Parameter count 2
3  Register count 0
4  Frame size 0
5  30 E> 0x1373c709b6 @ 0 : a5 00 00 00 StackCheck
6  56 S> 0x1373c709b7 @ 1 : 28 02 00 01
       ↪ LdaNamedProperty a0, [0], [1]
7  62 E> 0x1373c709bb @ 5 : 40 01 00 00 AddSmi [1],
       ↪ [0]
8  66 S> 0x1373c709be @ 8 : a9 00 00 00 Return
```

Figure 1. Example of a JavaScript function and its corresponding V8 bytecode instructions.

We have observed that V8 bytecode is resilient to script minification and static code-obfuscation techniques. Therefore, we believe that aligning such low-level representations could prove to be a useful aid in many program analysis tasks, such as code similarity study and malware analysis.

2.1.2 Non-Determinism in Browser Traces

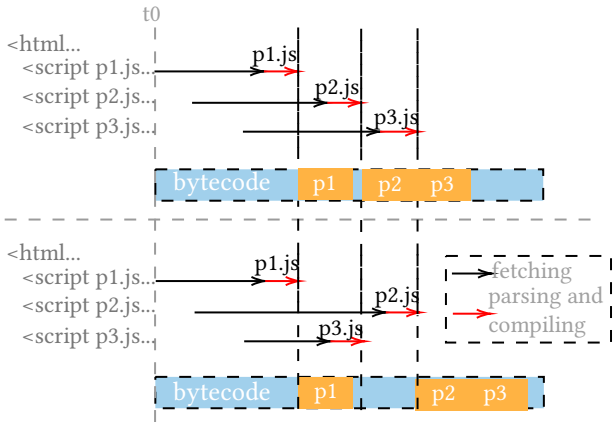

Figure 2. Illustration of two different script fetching and compiling traces for the same browser query.

Interestingly, browsers are fundamentally non deterministic, depending on web server availability, current workload,

and DNS caches through the network. Let us look at the example illustrated in Figure 2. It shows what happens when fetching a web page, which contains 3 scripts. The top and bottom parts illustrate, for the same page, two different executions. Dashed border rectangles represent complete bytecode generation traces. The blue spaces in the bar are V8 common builtin bytecode, which is systematically generated in all browser requests. Orange rectangles illustrate declared page scripts compilations. The complete bytecode trace is the union of both generated bytecodes, builtin V8 and page declared scripts. In the first case at the top of Figure 2, the scripts are fetched and compiled in the same order they are declared. In the second case, at the bottom, **p3.js** is carried and compiled first, before **p2.js** due to a possible network delay. However, V8's compiler will put all scripts compilations in the same order they are declared in the HTML page. The final result is two semantically equivalent bytecode compilations, where script blocks may not be strictly placed in the same position.

The slight differences that occur in the final bytecode for same browser queries motivate us to provide an efficient tool for traces alignment: traces where events occur in different orders but that have the same semantics must be considered as equivalent. The order of events should not confuse the trace comparison tool.

2.1.3 DTW Algorithm

The DTW algorithm has been introduced by Needleman and Wunsch for protein global alignment [18]. Global alignment means trace heads and tails are constrained to match each other in position. DTW is a popular technique for comparing traces in different domains, incl. software traces [14]. DTW finds the best global alignment between two traces, based on a generic similarity function between trace events and gaps.

Definition (Trace) A trace X is defined as a sequence of events. $X = x_1, x_2, ...x_N$ represents a trace of size N where each x_i is the event happening at the ith position.

Definition (Cost Matrix) D is a cost matrix for two traces X and Y of size n and m. D_{ij} stores the optimal cost alignment value for X and Y considered from the start up to the ith and jth positions respectively, that is the minimal cost of aligning x_i and y_j events at the same position in the final alignment.

The cost matrix is defined according to a distance function d and a gap cost γ as follows:

$$D_{0i} = \gamma * i$$

$$D_{j0} = \gamma * j$$

$$D_{ij} = min \begin{cases} D_{i-1j} + \gamma, \\ D_{ij-1} + \gamma \\ D_{i-1j-1} + d(x_i, y_j) \end{cases}$$

In every cell, the value D_{ij} is the minimum cost between putting a gap in one trace and the result of evaluating the distance function between events x_i and y_j.

Definition (Alignment Cost) Given two traces X and Y with sizes N and M respectively, the alignment cost is the value stored in D_{NM}.

Definition (Alignment Difficulty) Given two traces X and Y with sizes N and M respectively, the alignment difficulty is simply the multiplication of both sizes $N \times M$.

Definition (Warp Path) The warp path is the path to go from D_{NM} to the first element D_{00} minimizing the cumulative cost. In general more than one path may exist. Size of warp path is $O(N + M)$.

Definition (Aligned Trace) An aligned trace is a trace where the warp path is applied, i.e. some gaps have been put between some events in one of both traces.

In Figure 3 we illustrate the alignment between traces **abcababc** and **aabaca** with $\gamma = 1$, $d(x_i, y_j) = 2$ if $x_i \neq y_i$ and $d(x_i, y_j) = 0$ if $x_i = y_i$. The warp path is represented as the blue and orange lines going across the matrix from the top left corner to the bottom right corner. In this example, alignment cost is 4, as we can see in bottom right corner cell in Figure 3.

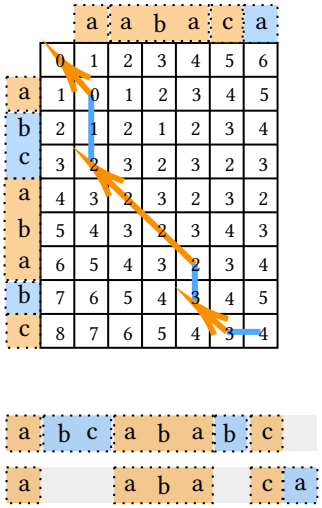

Figure 3. Cost matrix, warp path and applied alignment for **abcababc** and **aabaca** example traces.

3 STRAC: Trace Comparison Tool for V8

STRAC is an approach to compare large traces, tailored to bytecode traces of the V8 JavaScript engine. STRAC takes as input a trace of JavaScript V8 bytecode traces collected in the browser. It produces as output, a trace alignment, and a distance measure between the two traces. STRAC implements the DTW algorithm presented in Subsection 2.1.3. It is an open-source project publicly-available on https://github.com/KTH/STRAC . In this section, we explain the

key components and insights of STRAC to achieve scalable trace comparison.

3.1 Challenges Addressed by STRAC

Non-Determinism As shown in Subsection 2.1.2, V8 can provide two different bytecode traces for the same web page. In this case, both traces are semantically equivalent, but the global position of code modules can vary. These variations occur as a consequence of resource management, interpreter optimizations and JavaScript code fetching from the network. It is challenging because it can provide 1) false positives: two traces may be considered different even when they come from the same pages; 2) false negatives: two traces may be considered the same even when they come from two different pages.

Size Browser traces are huge and naive trace comparison fails on such traces because of memory requirements. For instance, aligning two traces of size 63137 and 58265 events requires a DTW cost matrix, represented as a bidimensional integer matrix, of 14.72 GB of memory. The challenge is to make trace comparison at the scale of browser traces, with tractable memory requirements.

3.2 DTW Distance Functions

The DTW algorithm has two main parameters: a distance function and a gap cost as explained in Subsection 2.1.3. The distance function between events affects the global alignment result, as we show in Subsection 4.5. It defines the matching of two different trace instructions if these instructions have a certain level of similarity. For example, when comparing *'AddSmi [0], [1]'* and *'AddSmi [1], [0]'* instructions, they can be considered as similar because the *AddSmi* operator is in both.

In STRAC, we define two distance functions for bytecode instructions.

$$d_{Sen}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ events are exactly} \\ \text{the same bytecode instruction} \\ c \text{ otherwise} \end{cases}$$

$$d_{Inst}(x_i, y_j) = \begin{cases} s \text{ if } x_i \text{ and } y_j \text{ bytecode instructions} \\ \text{share the same bytecode operator} \\ c \text{ otherwise} \end{cases}$$

Both require the identity relationship of the bytecode instruction. For V8 bytecode, based on our results (Subsection 4.5), it seems incoherent to accept an alignment match with two different elements instead of introducing the gap.

We now discuss the value of γ, s and c. The cost of introducing a gap, intuitively, must be less than the cost of matching two different events, i.e. $\gamma < s$. c is the value of matching two equal events, 0. The default values are based on our experience, $s = 5$, $\gamma = 1$ and $c = 0$. The three are configurable.

3.3 Buffering the Cost Matrix

The key limitation of DTW is the need for a large cost matrix to retrieve the warp path. Recall our example requiring 14.72 GB in Subsection 3.1. This means that a naive implementation can only compare small traces due to memory explosion.

In STRAC, we solve this problem by storing the cost matrix both in memory and disk. Only the appropriate values are kept in memory. Our key insight is that the current value D_{ij} in the cost matrix is calculated with the previous row and column, consequently, only $O(N)$ memory space is needed to compute D_{NM}. Thus, STRAC only maintains the current and previous row in memory for each DTW iteration. After processing a row, it is saved to disk. STRAC eventually saves the complete cost matrix to disk.

For traces with lengths 63137 and 58265, instead of 14.72 GB, STRAC requires no more than 86MB of memory for the trace alignment, which represents an improvement of 99.5% in memory consumption.

3.4 Retrieving the Warp Path

In addition to the alignment cost, it is necessary to obtain the warp path in order to create and analyze the aligned traces. Recall that the aligned traces are obtained by applying the warp path on both initial traces, as we mentioned in Subsection 2.1.3.

To retrieve the warp path from the final cost matrix, one goes backward and starts from the trace tail positions (D_{NM}). Cost matrix in D_{ij} depends on three neighbors D_{i-1j}, D_{ij-1} and D_{i-1j-1}. The backtracking process finishes when the trace start is reached, i.e. when the left top corner D_{00} is reached in the matrix. In the warp path construction process, trace indices are always decreasing by one, i.e. trace events are visited only once. Therefore, in STRAC, backtracking over the final cost matrix requires only $O(N + M)$ read operations on disk, which is scalable.

3.5 DTW Approximations

Due to the quadratic time and space complexity of DTW, previous work has proposed approximations to speed up the alignment process. STRAC also implements two state-of-the-art DTW approximations. We now mention these two approximations.

Fixed Regions Using fixed regions is a technique only to evaluate a specified region in the cost matrix [7, 12, 13, 24]. Consequently, the globally optimal warp path will not be found if it is not entirely in the window. This improvement speeds up DTW by a constant factor, but the execution time is still O(NM). STRAC provides support for fixed regions.

FastDTW [2] [25] is an approximation of DTW that has a linear time and space complexity. It combines data abstraction and constraint search in the solution space. STRAC implements FastDTW. Note that, for DTW and its approximations, the default mode is the buffering mode presented in Subsection 3.3.

3.6 Recapitulation

To sum up, STRAC is an optimized implementation of DTW and two approximations with distance functions dedicated to V8 bytecode traces and with neat handling of the cost matrix over memory and disk in order to scale.

4 Experimental Evaluation

We assess the scalability of STRAC for V8 bytecode trace comparison with the following research questions:

- RQ1 (Scalability): To what extent does STRAC scale to traces of real-world web pages?
- RQ2 (Consistency): To what extent does STRAC identify similarity in semantically-equivalent traces?
- RQ3 (Distance Functions): What is the effectiveness of STRAC support of different distance functions?

4.1 Study Subjects

Our experiment is based on tracing the home page of the following sites; google.com, github.com, wikipedia.org, youtube.com, four of the most visited websites, according to Alexa. We also add two sites based on personal interest: 2019.splashcon.org and kth.se, the homepage of our University. All those pages use JavaScript code. The traces were generated just opening the page without any other further action. Since the traces are non-deterministic, we collect 100 traces for the same page. This means we collect 600 traces in total.

Table 1. Descriptive statistics of our benchmark. The 6 sites are sorted by popularity according to the Alexa index. Example bytecodes are available in https://github.com/KTH/STRAC/tree/master/STRACAlign/src/test/resources/bytecodes.

Site	No. scripts	Bytecode size
google.com	5	85768
youtube.com	15	166626
wikipedia.org	4	48260
github.com	3	59384
kth.se	9	64178
2019.splashcon.org	17	147196

Table 1 gives an overview of the collected traces. The first column shows the real world website names. The second and third columns indicate the number of declared scripts and the bytecode size mean value (orange dots in Figure 4) respectively. For instance, Wikipedia loads 4 scripts and produces bytecode traces of 48260 bytecode instructions. This value is the lowest of our benchmark. On the contrary, for Youtube, the page declares 15 JavaScript scripts, and V8 generates traces of 166626 bytecode instructions, and this is due to the richer features of Youtube compared to Wikipedia. In our benchmark, the bytecode traces are in the range of 48k-166k instructions.

Recall that the bytecode traces are non-deterministic even for the same page (see Subsection 2.1.2). We measure how many instructions are contained in each V8 bytecode trace. Figure 4 illustrates the distribution of trace sizes as violin plots. This figure shows that there is a variance of bytecode traces for all pages (Wikipedia also has some variance but this is not shown in the figure because of the scale). This variance is a consequence of several stacked factors: resource management, interpreter optimization and JavaScript code fetching from the network. To our knowledge, this non-determinism in web traces is overlooked by research.

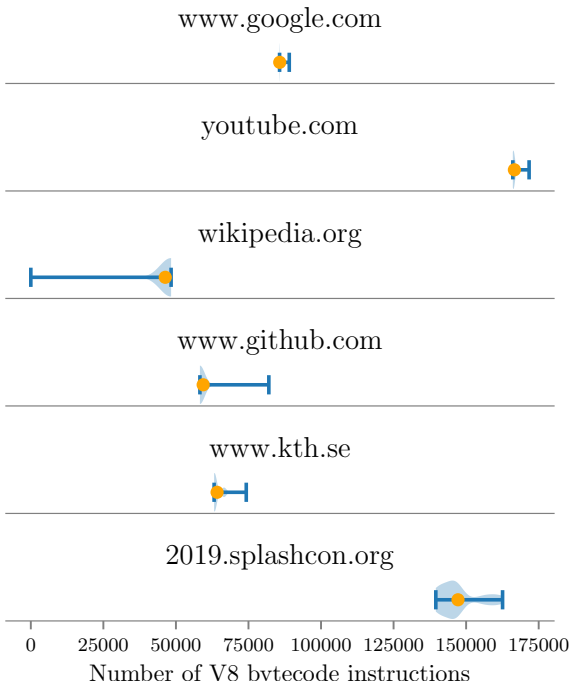

Figure 4. Variance of V8 bytecode trace size for 100 repetitions of the same query.

4.2 Experimental Methodology

Every trace is collected using a non-cached browser session, without plugins. This choice is motivated by two main reasons: 1) we have observed that cached scripts do not affect bytecode generation as direct network fetching does;

[2] The implementation mentioned in the original paper (https://cs.fit.edu/~pkc/FastDTW/) was not available at the moment of this work.

2) browser plugins are compiled to the same bytecode trace and in the scope of this work we are interested only in V8 bytecode traces directly generated from web page scripts.

To answer **RQ1**, we align 12 trace pairs randomly taken from the initial set of all possible trace pairs (600×600). We compare STRAC with different implementations of DTW 1) From public github repositories: rmaestre [3], dtaidistance [4] and pierre-rouanet [5]; 2) From R's dtw package [6] ; 3) The DTW implementation used in [15], slaypni [6]. For each comparison, we compute the average wall-clock execution time.

RQ2 is answered as follows. We select a random sample of 100 pairs from all possible trace pairs (600×600). We select 35 pairs of traces extracted from the same pages and 65 pairs of traces extracted from different pages. Alignment cost is measured for each pair using gap cost $\gamma = 1$ and event distance function d_{Sen} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We group and plot each pair alignment cost per site.

We answer **RQ3** using the same traces as *RQ2*. We compute DTW on each one of the 100 sampled pairs. We use the same gap cost $\gamma = 1$, but we compare the two distance functions d_{Sen} and d_{Inst} (defined in Subsection 3.2), with parameters: $s = 5$ and $c = 0$. We measure the alignment cost for each pair and compare the results with the ones obtained in *RQ2*.

The STRAC experimentation has been made on a PC with Intel Core i7 CPU and 16Gb DDR3 of RAM. We extract all traces from Chrome version 74.0.3729.169 (Official Build) (64-bit).

4.3 Answer to RQ1: Scalability

Figure 5 shows the execution time of 6 different alignment tools on 12 trace pairs. The X axis gives the size of the alignment problem, which is the multiplication of the size of both traces in number of bytecode instructions. The Y axis represents the execution time in seconds with a logarithmic scale.

First, we observe that four tools get out of memory for all the considered trace pairs: R-dtw, cpy-wannesm, rmaestre, cpy-slaypul (see the red dot in Figure 5). The main reason for this failure is that those tools need to store the cost matrix in memory. The least difficult trace comparison in the plot is a pair of traces of 48k instructions each. Finding the best alignment for this pair consists in analyzing an eight-bytes integer matrix of approx. 20GB (exactly 18632 millions of bytes). This memory requirement is almost the full memory of modern personal computers and it causes memory explosion at runtime. Applying the same analysis to the most difficult alignment in the plot shows requires 200GB of memory.

[3] https://github.com/rmaestre/FastDTW
[4] https://github.com/wannesm/dtaidistance
[5] https://github.com/pierre-rouanet/dtw
[6] https://github.com/slaypni/fastdtw

Second, py-wannesm and py-pierre-rouanet calculate the best alignment cost for the first 10 pairs, without any memory issue, even for problems in the order of magnitude close to 1.5×10^{10} in alignment difficulty. After this value, these tools also start to get memory issues for the same reason as the other tools. Yet, these succesfully align the 10 pairs (orange and green curves in Figure 5) thanks to an efficient use of Numpy [3] arrays to store cost matrix. Numpy arrays in Python are tailored to efficiently deal with arrays up to 20GB of memory in x64 architectures. We also observe that py-wannesm is always slower than py-pierre-rouanet. The main reason for this time difference is that py-wannesm does an extra pass through the cost matrix and py-pierre-rouanet does not do it.

Third, STRAC succesfully find the best alignment cost for all pairs in the benchmark, even for trace pairs that require memory beyond Numpy capabilities (the last two blue dots in Figure 5). The key insight behind is that STRAC implements the cost matrix data structure as a hybrid between memory and disk, i.e. moving such memory needs to disk.

Both Python implementations (py-wannesm and py-pierre-rouanet) systematically take at least one order of magnitude longer to run, compared to STRAC. The main reason behind this is that Python usually compiles code at runtime, while Java compiles it in advance, making a faster program. Besides, most JVMs perform Just-In-Time compilation to all or part of programs to native code, which significantly improves performance, but mainstream Python does not do this.

Recall that best alignment calculation using naive DTW implementation is non-scalable by its space-time quadratic nature, any implementation of DTW (even the one included in STRAC) eventually will run out of space (in memory or disk) and execution time will be near to impossible. However, STRAC can deal with all trace pairs of our benchmark thanks to its hybrid strategy that leverages both the disk and the memory. To align an average trace of 100k instructions, STRAC takes approx. 14 minutes in a PC like the one mentioned in Subsection 4.2.

4.4 Answer to RQ2: Consistency

In Figure 6, we plot the alignment cost for 100 trace pairs, the blue dots represent pairs extracted from the same page, the orange dots illustrate trace pairs taken from two different pages. Each column corresponds to a given web page. Green dots represent pairs with the maximum alignment cost for each site: an alignment of the web page treated in the column with a trace from the site cited above the dot. For example, the green dot in the first column is an alignment of a trace pair (2019.splashcon, youtube).

In Figure 6, we observe that, for each site, traces from the same page have a lower alignment cost. This is consistent with the fact that in these cases, the majority of both traces

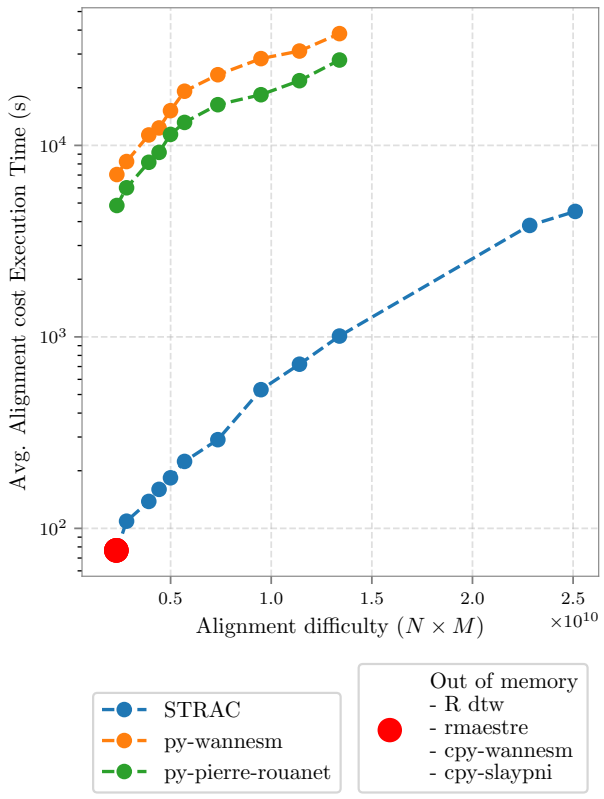

Figure 5. Execution time for **12** trace pair comparisons by 7 tools incl. STRAC. Y axis is in logarithmic scale. Four tools fail even on the smallest traces.

in the pair are the same. On the contrary, the alignment cost between traces from different pages is higher.

Some cases show blue dots with sparsed high values. This occurs when external scripts, declared in some pages, present a high variance in fetching process time. Also, it sometimes happens that for one script declared in a page, the remote servers sends different JavaScript code at each every request. Therefore, the generated bytecode varies more from one load to another, and the alignment cost is increased, showing a small margin between orange dots and the blue ones. However, we observe two scenarios when these phenomena are mitigated. First, when the bytecode generated from the external declaration is larger than the builtin bytecode (2019.splashcon, UNIV, and Youtube cases present a clear separation between clusters). Second, when the fetching process time is stable, as Wikipedia and Github cases show.

In the case of Google, we observe the worst possible scenario. This site has 5 external declared scripts (see Table 1), 3 of them have variable fetching time and their content varies at each load. These 3 scripts integrate Google Analytics features to the site. On the contrary, in the case of Wikipedia, external declared JavaScripts always provide the same code in almost constant time. As a result, the generated bytecode is more deterministic and alignment cost decreases for traces from the same site. In the case of Wikipedia, alignment costs for pairs of traces collected from the same page vary between 1926 and 2652. These values are the lowest alignment costs in the benchmark, and they differ from others in more than 2× in order of magnitude

Overall, the traces from the same (resp. different) page are located in separated clusters. In all cases, we also observe groups of orange dots that can be easily separated from other orange clusters. This separation is a consequence of semantic differences between sites and the increase of JavaScript declarations. For instance, in the first column of Figure 6, trace pairs from 2019.splashcon and Youtube home pages have higher alignment costs. This is a consequence of that Youtube is a richer feature site as 2019.splashcon is, but they semantically differ. We also observe this behavior in the case of Kth and Youtube trace pairs.

V8 compiles builtin JavaScript code to the same bytecode trace, as we discussed in Subsection 2.1.1. This bytecode generation is included in all collected traces. To validate this, we computed the V8 bytecode trace of an empty page: it contains 40k bytecode instructions on average. This also represents a constant noise in the alignment computation.

As Figure 6 illustrates, given the alignment cost of two semantically equivalent traces (blue dots) as a reference, STRAC is capable of identifying similarity with other page traces. However, we want to remark that STRAC accuracy gets improved when JavaScript declarations increase in the compared sites.

4.5 Answer to RQ3: Distance Functions

In Figure 7, we plot the alignment cost using distance d_{Ins}. Recall that d_{Ins} is less restrictive than d_{Sen}, the distance used to answer **RQ2**. By comparing Figure 7 and Figure 6, we observe interesting phenomena. First, changing the distance function breaks the clustering breakdown for Github, Google and Kth (some blue points get mixed with orange points). Second, the maximum alignment cost is lower than in Figure 6 for all sites. These phenomena are consequences of using a less restrictive distance function, i.e. with d_{Ins}, only the operator is analyzed in the bytecode instructions comparison. Overall, the choice of distance function matters. STRAC can be extended with new distance functions and provides d_{Sen} by default for properly aligning V8 bytecode traces.

We notice that the impact of the distance function is bigger for sites with less JavaScript. For Google, Github and Wikipedia, using d_{Ins} is bad because it breaks the clustering. For the remaining three websites, which involve more JavaScript features, while the alignment changes, the core property of the alignment of identifying semantically equivalent traces still holds.

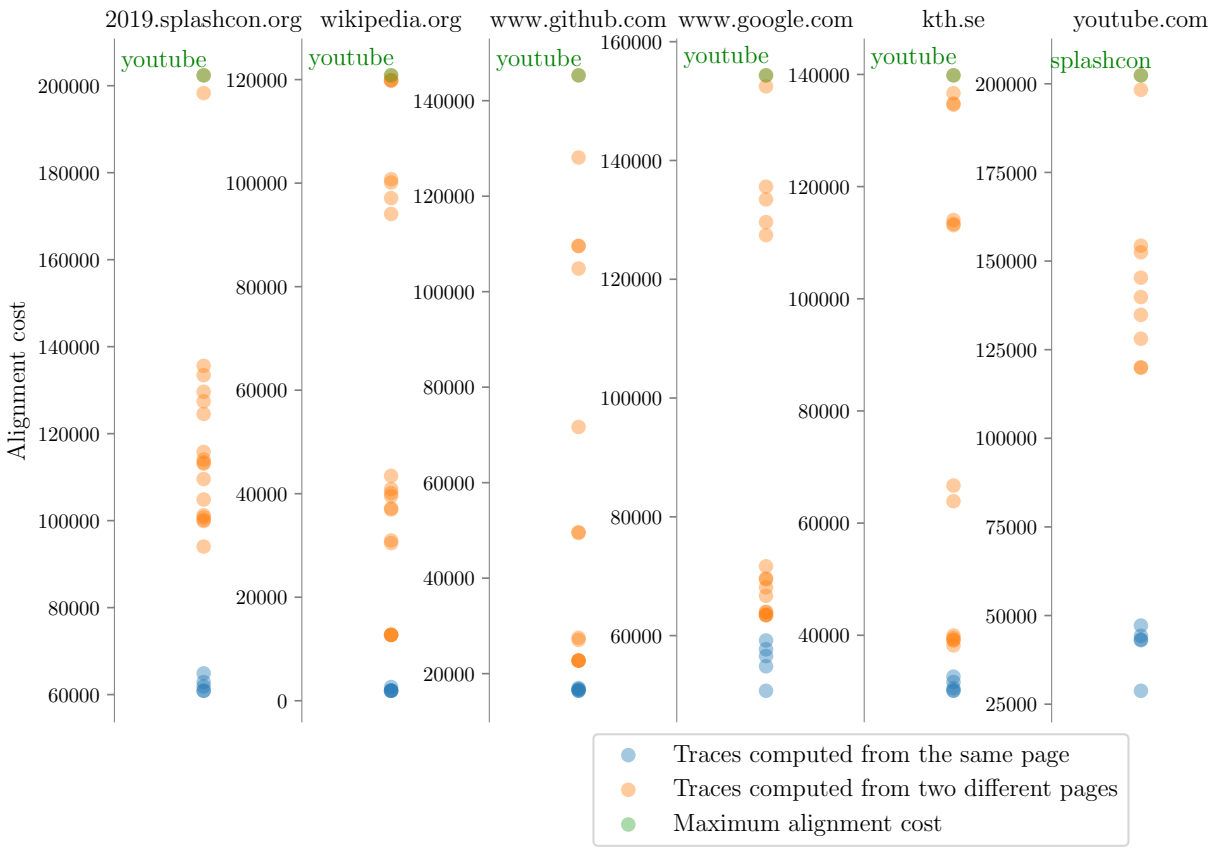

Figure 6. Alignment costs for 100 trace pair comparisons using d_{Sen} as distance function.

5 Related Work

DTW is memory greedy on trace size, a similar problem arises when dealing with streaming traces. Oregi et al. [20] and Martins et al. [15] present a generalization of DTW for large streaming data. They propose the use of incremental computation of the cost matrix complemented with a weighted event distance function adding event positions. However, their results may differ from the original DTW warp path. On the contrary, STRAC also computes the exact alignment cost without approximations.

Kargen et al. [10] propose a combination of data abstraction and FastDTW to align two program traces at the binary level. They record and analyze read and write operations to memory and x86 registers. Also, they argue and they show that their method scales to large traces. STRAC is also capable of analyzing such traces, but targets different kinds of traces: V8 bytecode traces, which are not handled by Kargen et al.

Ratanaworabhan et al. [23] instrument Internet Explorer to measure JavaScript runtime and static behavior in function calls and event handlers on real-world websites. By doing so, they show that common benchmarks, like SpiderMonkey and V8-Suite, are not representative of real application behavior. We could use STRAC to perform a similar analysis on modern browsers.

With JALANGI, Sen et al. [26] provide a framework to dynamically analyze JavaScript. The framework works through source code instrumentation. JALANGI associates shadow values to variables and objects in the instrumented code, Sen et al. argue that most of of state-of-the-art dynamic analysis techniques can be implemented, like concolic evaluation and taint analysis. However, JALANGI has several limitations dealing with builtin code and instrumentation can decrease instrumented code execution performance. With STRAC, we propose to use V8 bytecode traces to compare JavaScript semantic similarity without JavaScript instrumentation.

Fang et al. [5] propose a JavaScript malicious code detection model based on neural networks. To mitigate the obfuscation techniques used in malicious code, they analyze the dynamic information recorded in V8 bytecode traces. Both STRAC and Fang et al. consider V8 bytecode traces, yet the usages are different: they do anomaly detection while we do trace comparison.

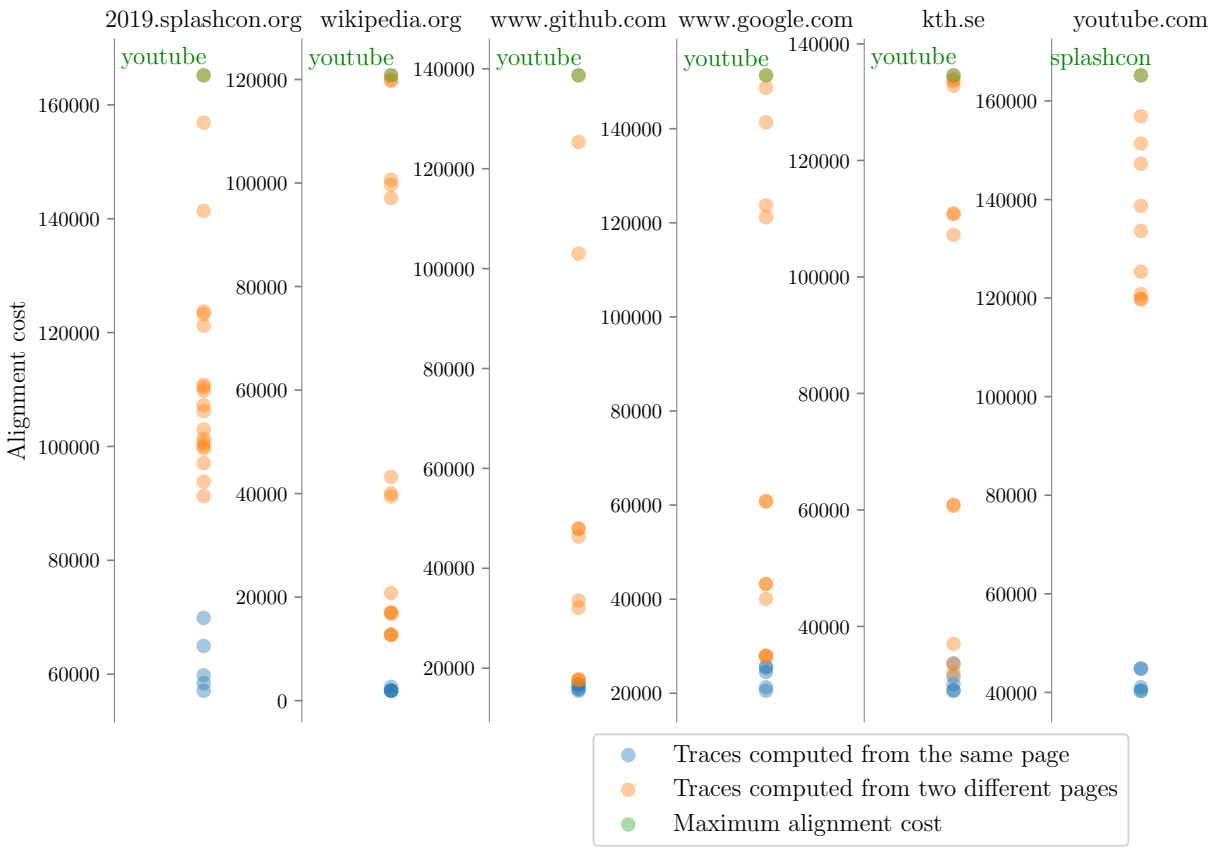

Figure 7. Alignment cost for 100 trace pair comparisons using d_{Ins} as distance function.

6 Conclusion

In this paper, we presented a tool, called STRAC, for aligning execution traces. STRAC is tailored to traces of the JavaScript V8 engine. STRAC implements an optimized version of the DTW algorithm and two of its approximations. Our experiments show that STRAC scales to real-world JavaScript traces consisting of V8 bytecodes. STRAC provides two distance functions for trace event comparison and can be configured with any arbitrary distance function. Our evaluation indicates that STRAC performs better than state of the art DTW implementations, for 6 representative web sites.

We have shown that V8 bytecode contains redundancy and that an empty page includes more than 40k trace instructions. By removing this redundant and useless trace instructions, the alignment would get better. In our future work, we will study how to remove redundancy in V8 bytecode traces, for providing a better behavioral similarity measure for modern web pages full of JavaScript code.

Acknowledgments

This material is based upon work supported by the Swedish Foundation for Strategic Research under the Trustfull project and by the Wallenberg Autonomous Systems and Software Program (WASP).

References

[1] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering - SIGSOFT/FSE '11* (2011). ACM Press, 267. https://doi.org/10.1145/2025113.2025151

[2] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

[3] Numpy community. 2018. Numeric python. https://www.numpy.org/index.html

[4] V8 JavaScript engine. 2016. *Ignition design documentation.* https://v8.dev/docs/ignition

[5] Y. Fang, C. Huang, L. Liu, and M. Xue. 2018. Research on Malicious JavaScript Detection Technology Based on LSTM. *IEEE Access* 6 (2018), 59118–59125. https://doi.org/10.1109/ACCESS.2018.2874098

[6] Toni Giorgino. 2009. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package. *Journal of Statistical Software, Articles* 31, 7 (2009), 1–24. https://doi.org/10.18637/jss.v031.i07

[7] F. Itakura. 1975. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 23, 1 (February 1975), 67–72. https://doi.org/10.1109/TASSP.1975.1162641

[8] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. 2007. Multiresolution Abnormal Trace Detection Using Varied-Length *n*-Grams and Automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37, 1 (Jan 2007), 86–97. https://doi.org/10.1109/TSMCC.2006.871569

[9] T. Kamiya. 2018. Code difference visualization by a call tree. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. 60–63. https://doi.org/10.1109/IWSC.2018.8327321

[10] Ulf Kargén and Nahid Shahmehri. 2017. Towards Robust Instruction-level Trace Alignment of Binary Code. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 342–352. http://dl.acm.org/citation.cfm?id=3155562.3155608

[11] Hyunjoo Kim, Jonghyun Kim, Youngsoo Kim, Ikkyun Kim, Kuinam J. Kim, and Hyuncheol Kim. 2017. Improvement of malware detection and classification using API call sequence alignment and visualization. *Cluster Computing* (12 Sep 2017). https://doi.org/10.1007/s10586-017-1110-2

[12] Daniel Lemire. 2008. Faster Retrieval with a Two-Pass Dynamic-Time-Warping Lower Bound. *CoRR* abs/0811.3301 (2008). arXiv:0811.3301 http://arxiv.org/abs/0811.3301

[13] Y. Lou, H. Ao, and Y. Dong. 2015. Improvement of Dynamic Time Warping (DTW) Algorithm. In *2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 384–387. https://doi.org/10.1109/DCABES.2015.103

[14] Marcelo De A. Maia, Victor Sobreira, Klérisson R. Paixão, Ra A. De Amo, and Ilmério R. Silva. 2008. Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*. 6–10.

[15] R. M. Martins and A. Kerren. 2018. Efficient Dynamic Time Warping for Big Data Streams. In *2018 IEEE International Conference on Big Data (Big Data)*. 2924–2929. https://doi.org/10.1109/BigData.2018.8621878

[16] Ross McIlroy. 2016. Ignition: V8 Interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit

[17] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 151–160. https://doi.org/10.1109/ICSME.2014.37

[18] Saul B. Needleman and Christian D. Wunsch. 1970. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[19] V8 official web page. 2019. *V8 JavaScript Engine*. https://v8.dev/

[20] Izaskun Oregi, Aritz Pérez, Javier Del Ser, and José A. Lozano. 2017. On-Line Dynamic Time Warping for Streaming Time Series. In *Machine Learning and Knowledge Discovery in Databases*, Michelangelo Ceci, Jaakko Hollmén, Ljupco Todorovski, and Saso Vens, Celinand Dzeroski (Eds.). Springer International Publishing, Cham, 591–605.

[21] The Chromium Projects. 2019. *Run Chromium with Flags - The Chromium Projects*. https://www.chromium.org/developers/how-tos/run-chromium-with-flags#TOC-V8-Flags

[22] David A Ramos and Dawson R. Engler. 2011. Practical, Low-effort Equivalence Verification of Real Code. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 669–685. http://dl.acm.org/citation.cfm?id=2032305.2032360

[23] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1863166.1863169

[24] H. Sakoe and S. Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 1 (February 1978), 43–49. https://doi.org/10.1109/TASSP.1978.1163055

[25] Stan Salvador and Philip Chan. 2007. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.* 11, 5 (Oct. 2007), 561–580. http://dl.acm.org/citation.cfm?id=1367985.1367993

[26] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[27] Ryo Suzuki, Gustavo Soares, Andrew Head, Elena Glassman, Ruan Reis, Melina Mongiovi, Loris D'Antoni, and Bjoern Hartmann. 2017. TraceDiff: Debugging Unexpected Code Behavior Using Trace Divergences. *CoRR* abs/1708.03786 (2017). arXiv:1708.03786 http://arxiv.org/abs/1708.03786

[28] Toon Verwaest and Marja Hölttä. 2019. *Blazingly Fast Parsing, Part 2: Lazy Parsing · V8*. https://v8.dev/blog/preparser

[29] M. Weber, R. Brendel, and H. Brunst. 2012. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. 247–254. https://doi.org/10.1109/ISPA.2012.40