The final publication is available at http://doi.org/10.1109/ICPP.2017.36

Parallel Construction of Simultaneous Deterministic Finite Automata on Shared-memory Multicores

Minyoung Jung*, Jinwoo Park*, Johann Blieberger† and Bernd Burgstaller*
*Yonsei University, Korea
Email: minyoung.jung@yonsei.ac.kr, bburg@cs.yonsei.ac.kr
†Vienna University of Technology, Austria
Email: blieb@auto.tuwien.ac.at

***Abstract*—String pattern matching with finite automata (FAs) is a well-established method across many areas in computer science. Until now, data dependencies inherent in the pattern matching algorithm have hampered effective parallelization. To overcome the dependency-constraint between subsequent matching steps, simultaneous deterministic finite automata (SFAs) have been recently introduced. Although an SFA facilitates parallel FA matching, SFA construction itself is limited by the exponential state-growth problem, which makes sequential SFA construction intractable for all but the smallest problem sizes.**

In this paper, we propose several optimizations to leverage parallelism, improve cache and memory utilization and greatly reduce the processing steps required to construct an SFA. We introduce fingerprints and hashing for efficient comparisons of SFA states. Kernels of x86 SIMD-instructions facilitate cache-locality and leverage data-parallelism with the construction of SFA states. Our parallelization for shared-memory multicores employs lock-free synchronization to minimize cache-coherence overhead. Our dynamic work-partitioning scheme employs work-stealing with thread-local work-queues. The structural properties of FAs allow efficient compression of SFA states. Our construction algorithm dynamically switches to in-memory compression of SFA states for problem sizes which approach the main memory size limit of a given system.

We evaluate our approach with patterns from the PROSITE protein database. We achieve speedups of up to 312x on a 64-core AMD system and 193x on a 44-core (88 hyperthreads) Intel system. Our SFA construction algorithm shows scalability on both evaluation platforms.

***Keywords*-SFA construction; fingerprints; hashing; work-stealing; lock-free programming; compression; parallelization; multicores;**

I. Introduction

The ubiquity of multicore architectures necessitates the adaptation of standard algorithms to expose parallelism and utilize parallel execution units. Parallelization improves performance and makes algorithms scalable to larger problem sizes. Searching a pattern from a large text is a prevalent processing step for various applications in computer science. Examples include text editors, compiler front-ends, scripting languages, web browsers, internet search engines, and security and DNA sequence analysis. Finite automata (FAs) derived from regular expressions enable such uses, but the underlying, sequential FA algorithm has linear complexity in the size of the input. Significant research effort has already been spent on parallelizing FA matching [2]–[12]. FA matching has been proven hard to be parallelized and inefficient on parallel architectures, due to the dependency between state transitions, i.e., for an FA to perform a state transition, the result-state from the previous transition must be known.

To speed up FA matching on parallel architectures, simultaneous deterministic finite automata (SFAs) have been proposed [13]. Given an FA A with n states, the corresponding SFA $S(A)$ simulates n parallel instances of FA A. (This simulation is similar to the simulation of a non-deterministic FA by a deterministic FA through the subset construction algorithm [14].) In particular, an SFA state s is a vector of dimension n of FA states; the SFA start-state is the vector $\langle q_0, \ldots, q_{n-1} \rangle$, where each q_i is a state of the original FA. An SFA transition from SFA state s_1 to SFA state s_2 on input symbol σ, denoted by $s_1 \xrightarrow{\sigma} s_2$, subsumes the transitions of the underlying FA from each of the FA states in vector s_1. Running the SFA on a sub-string of the input constitutes n parallel executions of the corresponding FA, each from a unique state from the set of A's states. Given an SFA, it is then possible to split the input into sub-strings, match each of the sub-strings in parallel with the SFA, and combine the result vectors by reduction.

We identify the representation and comparison of SFA states as the most resource-consuming part of the SFA construction algorithm. Because of the exponential state-growth problem, an FA of size n may result in $\mathcal{O}(n^n)$ SFA states. These space requirements and the proportional amount of processing steps make sequential SFA construction intractable for real-world problem sizes. E.g., we observed patterns from the PROSITE protein sequence database [15], [16] to exhibit SFA construction times of several hours before running out of memory on a contemporary Intel Xeon E5-2699 server with 512 GB of main memory.

To mitigate the space constraints of SFA construction, we employ in-memory compression of SFA states. In-memory compression is initiated once our construction method is about to deplete the overall system memory. SFA construction is suspended, all prior-constructed SFA states are

compressed, and SFA construction resumes by applying compression to all subsequently generated SFA states. Because of this three-phase algorithm, we are able to avoid the compression overhead for smaller problem sizes for which the exhaustive state representation fits in memory. Because of structural properties of the underlying FAs, dictionary-based data-compression is effective with SFA states, yielding typical compression ratios of 8x–18x, with FAs of particular patterns reaching 133x.

Because of its high memory requirements, SFA construction suffers from poor cache-locality, which severely impacts performance. We introduce fingerprints as a compact "shorthand" representation for SFA states. Employing fingerprints instead of the entire, byte-by-byte representation improves the efficiency of SFA state-comparisons, which is a frequent operation with SFA construction. The compact size of fingerprints vastly improves cache-locality of the SFA construction algorithm. Note that mapping the in-memory representation of an SFA state to a fingerprint is a surjective operation, i.e., two distinct SFA states may be mapped onto the same fingerprint. Therefore, for identical fingerprints our algorithm has to resort to the exhaustive, byte-by-byte comparison. Because this case is less frequent, it does not impose a performance problem in practice. Nevertheless, our proposed construction algorithm has to maintain both the fingerprint and the exhaustive SFA state representation in memory.

We introduce a series of kernels of x86 SIMD-instructions to facilitate cache-locality and leverage data-parallelism with the construction of SFA states. By introducing hashing, we reduce the look-up of SFA states in memory to $\mathcal{O}(1)$. Our hash function uses the fingerprint of an SFA-state as the key. We designed our own thread-local work-queues to support efficient work-stealing with our dynamic work partitioning scheme. Our parallelization for shared-memory multicores employs lock-free synchronization to minimize cache-coherence overhead.

This paper makes the following contributions:

1) We introduce fingerprints to speed up comparisons of SFA-states and improve the cache-locality of the SFA construction algorithm. Fingerprint-based hashing of SFA states enables us to decide the set-membership of SFA states and perform look-ups with a time-complexity of $\mathcal{O}(1)$.
2) The construction of SFA states is a data-parallel operation, for which we provide a series of kernels based on x86 SIMD intrinsics. Besides leveraging data-parallelism, our kernels improve the cache-locality of SFA state construction.
3) We perform in-memory compression of SFA states to mitigate the space constraints of large problem sizes. We motivate the effectiveness of dictionary-based compression with SFA states to allow larger problem sizes.
4) We parallelize SFA construction for shared-memory multicores. Our thread-local queues support work-stealing and dynamic work partitioning among threads. We minimize the cache-coherence overhead by using lock-free synchronization on all employed data-structures, including our thread-local work-queues and the hash-table of SFA states.
5) We conduct SFA construction in three phases, to optimize performance to the extend that an SFA fits into main memory. A compression phase shrinks the in-memory SFA state representation once a critical memory threshold is reached. Construction is then resumed on the compressed state representation. All three phases are jointly conducted by all worker threads.
6) We evaluate our SFA construction algorithm and SFA-based FA matching for a comprehensive selection of patterns from the PROSITE protein sequence database [15], [16]. Evaluation has been conducted on a 4-CPU (64 cores) AMD Opteron system and a 2-CPU (44 cores, 2 hyperthreads per core) Intel Broadwell E5-2699 v4 system.

The remainder of this paper is organized as follows. In Section II we explain the relevant background material, specifically the sequential SFA construction method. In Section III, we present our optimization methods for the SFA construction algorithm. Section IV contains our experimental evaluation. We discuss the related work in Section V and draw our conclusions in Section VI.

II. Background

Fingerprints: Fingerprints are short bit-strings for larger objects. If two fingerprints are different, the corresponding objects are known to be different. There is a small probability that two objects map to the same fingerprint, which is called a fingerprint collision. There exists a large body of work on fingerprints and their associated hash-functions. Michael O. Rabin's method [17], [18] creates fingerprints from arbitrary bit-strings by interpreting them as a polynomial in $\mathbb{Z}_2$. More recent hash-functions include CityHash [19], and FarmHash [20].

Finite Automata: A tuple $(Q, \Sigma, \delta, I, F)$ describes a deterministic finite automaton (DFA) A. Q is a finite set of states and $|Q|$ is referred to as the size of the DFA. Σ is a finite alphabet of characters and Σ^* is the set of strings over Σ. $I \subseteq Q$ is a set of initial states, but with DFAs there is one initial state $q_0 \in Q$, called the start state. $F \subseteq Q$ is the set of accepting states. δ is a transition function of $Q \times \Sigma \rightarrow Q$. Transition function δ is extended to δ^*: $\delta^*(q, va) = p \Leftrightarrow \delta^*(q, v) = q', \delta(q', a) = p, a \in \Sigma, v \in \Sigma^*$. An input string *Str* over Σ is accepted by DFA A if the DFA contains a labeled path from q_0 to a final state such that this path reads *Str*. The DFA membership test is conducted by computing $\delta^*(q_0, Str)$ and checking whether the result is an

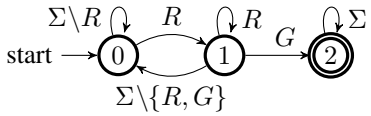
accepting state. The symbol in the x th position of the input string is denoted by $Str[x]$.

Fig. 1a shows an example FA over the alphabet of one-letter abbreviations for the 20 amino-acids ($\Sigma = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$). The FA with start state 0 and final state 2 accepts input strings that contain the sequence RG. The table in Fig. 1b encodes the transition function δ, and Fig. 1c shows a sequential matching routine.

It is worth noting that the FA in Fig. 1a accepts if the searched pattern (RG) occurs as a substring at *any* position in the input. This constitutes a catenation of three FAs corresponding to the string-sets for Σ^*, RE and Σ^*. The catenation operation has exponential state complexity, i.e., the catenation of an m-state and an n-state FA has up to $m2^n - 2^{n-1}$ states [1, Theorem 2.1]. Throughout this paper—unless stated otherwise—we have applied the aforementioned catenation operations on all FAs to enable pattern-matching at any position in the input.

SFA Construction: Algorithm 1 denotes the sequential algorithm proposed in [13] to construct an SFA $\mathcal{S}$ from an FA $\mathcal{A}$. The SFA start-state, denoted by the state-mapping f_I, constitutes a vector $\langle q_0, \ldots, q_{n-1} \rangle$, where each q_i is a state of the original FA. Beginning with the start-state, the algorithm simulates the entire SFA by generating new SFA-states, checking whether they are already known, and adding new states to the state-set Q_s. The SFA's transition function δ_s is extended incrementally with each new SFA state. (Details of the algorithm can be found in [13].)

Fig. 2 depicts the SFA constructed from the FA in Fig. 1. SFA states Q_s constitute state-mappings f_i, $0 \leq i \leq 5$, as depicted in the state mapping table.



(a) Example FA

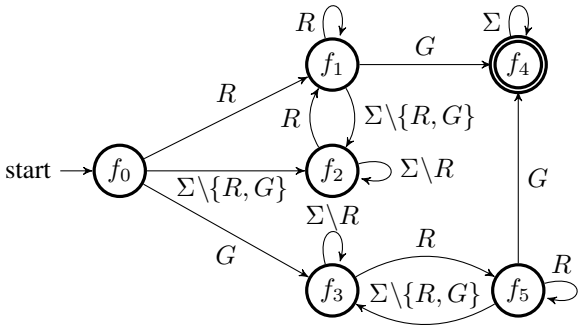
δ	$\Sigma\backslash\{R,G\}$	R	G
0	0	1	0
1	0	1	2
2	2	2	2

(b) Transition table

```
1 state ← 0
2 for i ← 0 to |Str| − 1 do
3     state ← δ(state, Str[i])
```

(c) Sequential matching routine on input *Str*

Figure 1: Example FA: state diagram, transition table and sequential matching routine



(a) Example SFA state diagram

f_0	$0 \to \{0\}$ $1 \to \{1\}$ $2 \to \{2\}$	f_1	$0 \to \{1\}$ $1 \to \{1\}$ $2 \to \{2\}$	f_2	$0 \to \{0\}$ $1 \to \{0\}$ $2 \to \{2\}$
f_3	$0 \to \{0\}$ $1 \to \{2\}$ $2 \to \{2\}$	f_4	$0 \to \{2\}$ $1 \to \{2\}$ $2 \to \{2\}$	f_5	$0 \to \{1\}$ $1 \to \{2\}$ $2 \to \{2\}$

(b) Example SFA state mapping table

Figure 2: Example SFA constructed from the FA in Fig. 1

III. Optimizing the SFA Construction Algorithm

The time complexity of Algorithm 1 can be determined as follows: The outermost while-loop (line 2) iterates until the set Q_tmp containing generated but non-processed SFA states is empty, i.e., $|Q_\mathrm{s}|$ times, once per SFA state (equal to the size of the resulting SFA). In the loop body, the successor states of each SFA state are calculated on each symbol, one by one, in the for-loop starting at line 5. Thus the for-loop iterates $|\Sigma|$ times per SFA state. To decide whether a newly created state is already in the set Q_s, the "exhaustive" set membership test (comparing all FA states in an SFA state)

Algorithm 1: SFA Construction

Require: Automaton $\mathcal{A} = (Q, \Sigma, \delta, I, F)$
Ensure : SFA $\mathcal{S} = (Q_\mathrm{s}, \Sigma, \delta_\mathrm{s}, I_\mathrm{s}, F_\mathrm{s})$ is equivalent to automaton $\mathcal{A}$

1 $Q_\mathrm{s} \leftarrow \emptyset, Q_\mathrm{tmp} \leftarrow \{f_\mathrm{I}\}$
2 **while** $Q_\mathrm{tmp} \neq \emptyset$ **do**
3 choose and remove a mapping f from Q_tmp
4 $Q_\mathrm{s} \leftarrow Q_\mathrm{s} \cup \{f\}$
5 **forall the** $\sigma \in \Sigma$ **do**
6 $q \in Q \quad f_\mathrm{next}(q) := \bigcup_{q' \in f(q)} \delta(q', \sigma)$
7 $\delta_\mathrm{s}[f, \sigma] \leftarrow f_\mathrm{next}$
8 **if** $f_\mathrm{next} \notin Q_\mathrm{s}, Q_\mathrm{tmp}$ **then**
9 $Q_\mathrm{tmp} \leftarrow Q_\mathrm{tmp} \cup \{f_\mathrm{next}\}$
10 $I_\mathrm{s} \leftarrow \{f_\mathrm{I}\}$
11 $F_\mathrm{s} \leftarrow \{f \in Q_\mathrm{s} | \exists q \in I | f(q) \cap F \neq \emptyset\}$

between the new state and all other existing states must be conducted. If we assume the worst-case, i.e., all SFA states are different, this comparison will have to be done for each state in Q_s and Q_tmp and each membership test requires at most $|Q|$ comparisons of FA states. The time complexity of this algorithm is as follows.

$$\mathcal{O}\Big(\sum_{i=1}^{|Q_s|}\sum_{j=1}^{|\Sigma|}(|Q|+|Q|\times i)\Big) = \\ = \mathcal{O}\Big(\frac{1}{2}\times|\Sigma|\times|Q|\times|Q_s|\times(|Q_s|+3)\Big) \tag{1}$$

From Equation (1) it is evident that the number of generated SFA states is the most significant contributing factor for the running-time of this algorithm. Comparisons for newly generated SFA states take the majority of time, i.e., $\mathcal{O}(|\Sigma|\times|Q|\times|Q_\mathrm{s}|\times(|Q_\mathrm{s}|+1)\times\frac{1}{2})$. Checking termination i.e., $\mathcal{O}(|Q_\mathrm{s}|)$, and state transitions, i.e., $\mathcal{O}(|Q_\mathrm{s}|\times|\Sigma|\times|Q|)$, only take a small portion of the algorithm's running-time. Consequently, decreasing the time taken by state comparisons is a major goal of the following optimization.

A. *Fingerprints and Hashing*

Computing a fingerprint from the binary representation of an SFA state allows us to speed up SFA state-comparisons to $\mathcal{O}(1)$ for a pair of distinct SFA states. Only when the fingerprints of two SFA states are equal, our comparison must resort to the exhaustive, byte-by-byte comparison. This follows from the surjective nature of hash-functions, which potentially map two distinct SFA states to the same fingerprint. We created our own SIMD-based implementation of Rabin fingerprints [17], [18], which creates fingerprints from arbitrary bit-strings by interpreting them as a polynomial in $\mathbb{Z}_2$. Our implementation specifically employed the `PCLMULQDQ` SSE instruction to speed up carry-less polynomial multiplication. On both of our evaluation platforms (see Table I), this implementation created fingerprints for SFA-states at a rate of 1.1 bytes per CPU-cycle. In comparison, FarmHash [20], and CityHash [19] created fingerprints at higher rates. For CityHash, the fastest hash-implementation in our survey, the measured rate was 5.1 bytes per CPU-cycle. The reason for the lesser performance of Rabin fingerprints is likely the `PCLMULQDQ` SSE instruction, which has a relatively high latency (7) and low throughput (2) on recent x86 architectures [21]. The second criterion for choosing a hash function was the number of collisions, for which we did not experience a significant difference between CityHash and Rabin's method. We thus chose CityHash as the hash-function for computing fingerprints of SFA states. However, it should be mentioned that for a probabilistic version of our algorithm, which would store fingerprints only, Rabin fingerprints would be the better choice, because Rabin's method is capable of providing tight bounds on the number of expected hash-collisions, and it allows to adjust the collision rate through the degree of the polynomials in $\mathbb{Z}_2$ (see [18]). For this paper, we did not investigate a probabilistic implementation.

Table I: Hardware specifications. "HT" denotes the number of hyper-threads per CPU.

Name	CPU Model	CPUs	$\frac{\text{Cores (HTs)}}{\text{CPU}}$	Main Mem.
Intel	Xeon E5-2699 v4	2	22 (44)	512 GB
AMD	Opteron 6378	4	16	128 GB

By exploiting fingerprints further, we introduce a hash-table data structure to our construction algorithm. Because fingerprints are stored in an `unsigned int` or `unsigned long long` data type (`uint32_t` or `uint64_t`), we apply them as the keys (modulo the size of the hash-table) for hashing of fingerprints. This leaves us with fingerprint-collisions (from fingerprinting) and hash-collisions due to the modulo-operation that maps keys to our hash-table. Our hash-table implementation thus must allow duplicated keys. We store duplicated keys by chaining with linked lists. Open addressing, which resolves collisions by probing, impacts performance because of the search for alternative free slots when fingerprint-collisions occur. In particular, searching is required whenever we add a state to the hash-table or lookup the hash-table to check whether a generated state was already stored. For large DFAs with several thousand DFA states, the size of an SFA state grows to several kilobytes of data. Each SFA state contains the state's fingerprint, the byte-by-byte binary representation, and a pointer used for chaining the SFAs that result from fingerprint- and hash-collisions.

During SFA construction, a fingerprint is computed for each newly-generated SFA-state. The fingerprint indexes into the hash-table. If there is no collision, the generated state is added to the hash-table at the entry of the fingerprint's hash index. Otherwise, the linked list of SFA-states is traversed and compared one by one. Note that the SFA-states in the list will have different fingerprints than the to-be-added state in case of hash-collisions, and the same fingerprint in case of fingerprint-collisions or if the compared states are identical. If the comparison detects identical fingerprints, the exhaustive, byte-by-byte comparison will be performed. In the best case, we only need $\mathcal{O}(1)$ time to find states that have the same fingerprint.

As an example, suppose our algorithm is going to derive new SFA states from a given SFA state s_0 on a set of input symbols $\Sigma = \{\delta_0, \delta_1, \delta_2\}$. We transpose the transition table δ according to the DFA states in s_0. In this example, we assume that the SFA-state s_0 consists of the DFA-states $\{q_1, q_0, q_2\}$. To compute the SFA states derived from state s_0, each row in the transition table corresponding to DFA states q_1, q_0 and q_2 becomes one column in the transposed transition table. Thus, with this transposition

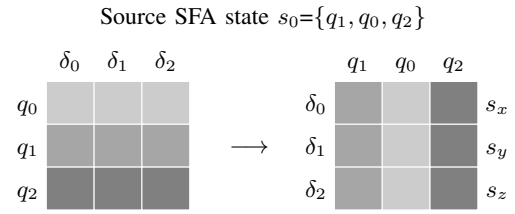


Figure 3: Example parameterized transposition

operation, the source SFA state (s_0 in this example) parameterizes the transposition in selecting the rows to be transposed. As depicted in the right part of Fig. 3, the derived SFA states s_x, s_y and s_z each occupy one row in the transposed table.

As follows from the left part of Fig. 3, the transition table is read in row-major order, with each row corresponding to one DFA state in s_0. Each such row becomes one column in the transposed table in the right part of Fig. 3. Each *row* in the transposed table represents a new SFA state (which will again be processed in row-major order, to facilitate locality).

The sequential algorithm for this approach is the same as Algorithm 1, except for line 6, which is replaced by the parameterized transposition.

To speed up this process, we created a series of x86 SIMD-intrinsics-based kernels to leverage the data-parallelism contained in this operation. We devised an 8×8 kernel for 32 bit data (based on 256 bit AVX2 instructions), and an 8×8 and an 8×4 kernel for 16 bit data (based on SSE 128 bit instructions). We implemented a 16×16 kernel for 16 bit data (based on 256 bit AVX2 instructions), but four 8×8 kernels showed slightly higher speedup than one 16×16 kernel. Each of these x×y kernels takes x rows of y elements and transposes them into y rows of x elements. As mentioned above, the input rows are determined by the DFA states of the source SFA state (parameterized transposition).

B. Parallelization for Multicore Architectures

1) Sources of Parallelism: Similar to parallel subset construction [22], the sequential SFA construction algorithm (Algorithm 1) contains sources of coarse-grained, medium-grained and fine-grained parallelism.

We found fine-grained parallelism to be impractical for the following reasons: (1) dividing across state transitions (line 6) of FA states yields not enough work, because the total amount of work for state transitions is rather small ($|Q_s| \times |\Sigma| \times |Q|$) compared to the state comparison part. It (2) requires a non-negligible amount of communication to assign and distribute work without leaving processors idle (in particular when the number of processors exceeds the number of FA states), and (3) gathering of partial results from individual transitions to construct an SFA state incurs further synchronization overhead between threads. Although fine-grained parallelization benefits from cache locality (transitions are split based on the FA states and one row in the transition table δ corresponds to the transition information of one FA state), the communication and synchronization overhead outweighs all possible gains.

The most significant and easy to utilize source of parallelism is coarse-grained parallelism, which partitions SFA states among workers. In this case, the while loop (from lines 2 to 9 in Algorithm 1) is parallelized. However, this parallelization is only effective if there are enough unprocessed states in Q_{tmp} such that no worker is left idle. This approach is thus effective for FAs which have a large number of states.

For medium-grained parallelism, we assign work based on symbols from alphabet Σ. Each thread can work independently to produce new SFA states on transitions from the given symbols, test for possible duplication of SFA states and add SFA states to the set Q_{tmp}. Although threads can work independently, this scheme still requires a work partition that distributes work evenly to processors. Special considerations need to be taken when the number of processors is larger than the number of symbols in Σ.

We utilize coarse-grained parallelism with the transposed transition table and medium-grained parallelism for work-stealing.

2) Work-Stealing: In parallel computing, load balancing is considered as one of the most significant factors to improve performance. Load balancing strategies are classified into static and dynamic schemes. If the workload required from each thread is predictable, allocating work statically will show higher speedups, because overhead to decide the workload at runtime is avoided. If the workload of each thread is difficult to anticipate, allocating work dynamically might achieve higher speedups, because threads which finished their work earlier can receive more work rather than wait for their peers to finish.

Work-stealing is an efficient dynamic work distribution method, because it requires a runtime decision only when a thread runs out of work. Because the SFA construction algorithm is an algorithm which does not allow to anticipate the workload for each thread a-priory, we apply work-stealing with our parallelization of the SFA construction algorithm. In our implementation, a single SFA state constitutes one work-item. New SFA states are generating during SFA construction; each thread has a local queue to store the SFA states it generated. To obtain work, the owner thread will consult its own local queue first. If a thread's local queue is empty, the thread will steal work from other threads' queues.

In the initial stage of SFA construction, only the start-state f_{I} is available. If threads are allowed to steal work from the beginning, all threads but the thread working on the start-state become thieves, which deteriorates performance because of competition. Our construction algorithm combines static work distribution with a single global queue

and dynamic work distribution (work-stealing) with thread-local queues. In the initial stage of the SFA construction algorithm, threads will work on a single global queue. After a threshold number of SFA states have been generated, threads switch to their own local work queues. With small benchmarks, the global queue will suffice to generate the entire SFA.

With our global queue, work is statically allocated: threads use their thread ID to index into the queue and de-queue work from the front. To en-queue work, threads use a CAS operation to synchronize on the current back-position of the queue.

During work-stealing from thread-local queues, a CAS operation is required to avoid the situation that several thieves de-queue the same SFA state. If a thief fails to steal a state, it searches for another state. To reduce cache coherence overhead, a thief starts to search a state from the closest queue, i.e., a queue whose owner thread shares its cache with the thief.

C. In-memory Compression of SFA states

To mitigate the state explosion problem and increase the problem sizes that can fit into a computer's main memory, we apply in-memory compression of SFA states. Data-compression is effective with SFA states, because the distribution of FA states of a given SFA state is often dominated by only a few FA states. This observation is related to structural properties of FAs, in particular that some FA states are more likely to occur during matching. E.g., FA states that are part of a cycle in the FA state diagram are likely to be reached frequently during matching. The space requirements for SFA states dominate the working set of the SFA construction algorithm, which means that shrinking the representation of SFA states significantly increases the computable problem sizes.

We conducted an experiment where we compressed the in-memory representation of SFA states to determine their potential for compression. We selected three PROSITE SFAs and the r500 SFA and extracted the binary in-memory representation of 10 SFA states each. SFA states were selected from aequidistant positions as they were computed by the sequential SFA construction algorithm. We employed the Squash compression benchmark [23], which is a collection of 43 commonly used codecs including deflate, zip, gzip and bzip2. Each of the 43 codecs was applied on the SFA state-sample, and the achieved compression ratios and execution times were obtained. We found LZ77-based codecs, in particular the *deflate* codec, to achieve the highest compression ratios, typically between 17x–30x. (The compression ratio was computed by dividing the uncompressed size by the compressed size.) Note that these compression ratios are high, compared to corpora such as English text or Wikipedia contents, which rarely exceed a ratio of 5x [23]. The r500 SFA is a synthetic benchmark [13], and it does not employ the catenation explained in Sect. I. For this type of benchmark, the deflate codec achieved a compression ratio of 95x, because SFA states are largely dominated by the FA's error (sink) state. For this type of SFA, run-length encoding will be able to produce similar results, but we did not pursue this direction as in our experiments we allow patterns to match at every position in the input.

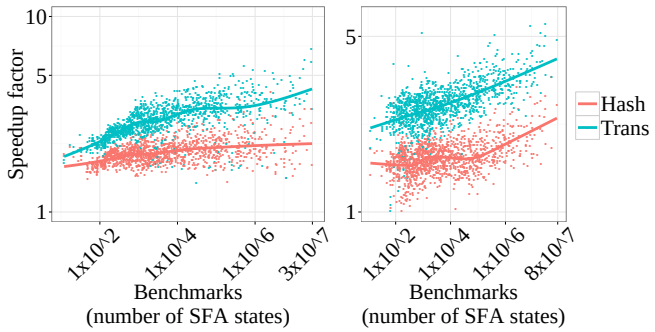
Compression is a costly operation and decompression of the unprocessed SFA states is necessary whenever they are processed because of state transitions. Our construction algorithm thus initially stores SFA states without compression. Thereby performance is optimized for problem sizes that fit into the main memory. Once our memory manager detects that the overall memory usage exceeds a critical threshold, it flags the start of our algorithm's compression phase. Workers will be in different parts of the construction algorithm and we thus additionally use one flag for each worker to confirm entering the compression phase. Workers which detected the memory manager's start signal immediately start compression, but uncompressed SFA states can only be reclaimed by the memory manager once all threads confirmed to be in the compression phase. Because the compressed representation of an SFA state will be stored in a different location in memory, the compression phase entails a re-build of our hash-table, i.e., at the beginning of the compression phase the hash-table is emptied, and workers enter the compressed states. (There is no need to check for duplicate states with this operation, which improves efficiency.) After all states have been compressed, workers re-sume SFA construction in "compression-mode", i.e., by compressing each newly-found state.

We found the deflate codec to be one order of magnitude slower than a plain memory copy operation, which we deemed reasonable, given that once the system is about to run out of memory, tractability takes priority over performance.

IV. Experimental Results

In our experimental evaluation, we demonstrate how all optimizations presented in Section III affect the performance of the SFA construction algorithm. We evaluated SFA construction and matching on one AMD and one Intel system characterized in Table I. Both systems were running Linux CentOS version 7. POSIX threads [24] were used to parallelize SFA construction and matching across multiple cores. We use Grail+ [25], [26] to generate minimal DFAs from regular expressions. Our framework reads DFAs and input strings in Grail+ format and converts them to its own internal representation.

For our experimental evaluation, we chose 1250 patterns from the PROSITE protein database [15], excluding those patterns which took more than several hours to compute with the sequential baseline. The smallest benchmark in our sample consists of 5 DFA states, the largest one consists of



(a) on the AMD system (b) on the Intel system

Figure 4: Performance of hashing and parameterized transposition over the sequential base-line.

7235 DFA states. We experimented with the r500 pattern, which is a synthetic benchmark from [13].

Performance figures were obtained from hardware performance counters using PAPI [27]. On the AMD system, the clock frequency was set to the nominal maximum frequency, i.e., 2.40 GHz. AMD turbo core was enabled in the BIOS. The Intel system employed the p-state driver. We used the pstate-frequency utility to set the "max" power plan, which requests the CPU's highest performance level and enables Intel turbo-boost. With this setting, the CPU clock frequency of an individual core ranges from 2.80 GHz ($\geq$9 active cores per CPU) up to 3.60 GHz ($\leq$2 active cores per CPU).

We ran the entire sequential experiment only once because one run including the baseline, hashing with fingerprints and parameterized transposition with hashing takes already several days to complete. We ran the entire parallelization experiment three times and took the median execution time. Execution times were the times of the longest-running thread. Execution times were stable, with the average coefficient of variation being 8.6%. The interpolated lines of diagrams were created using R's local regression method.

A. Optimizations for the Sequential Algorithm

Fig. 4 illustrates the speedups achieved by hashing and parameterized transposition over the non-optimized base-line; both axes use a logarithmic scale. The non-optimized baseline employs a C++ STL map based on red-black trees, following the sequential SFA construction method from [13], [28]. Hashing describes the optimization of hashing with fingerprints and parameterized transposition describes the optimization of hashing and parameterized transposition of transition tables (i.e., our most efficient method that does not use multiple threads).

The maximum speedups achieved by hashing on the AMD and the Intel system are 4.1x and 3.1x, with median speedups of 2.0x and 1.7x, respectively. The maximum speedups accomplished by the combination of hashing and parameterized transposition on the AMD and the Intel system are 6.8x and 5.2x, whereas the median speedups of them are 2.9x and 2.8x. The execution time of our most efficient sequential method will be the baseline for the speedup achieved by parallelization. On the Intel system, our implementation of baseline, hashing and parameterized transposition resulted in execution times of 36.6 s, 10.6 s and 6.4 s with the r500 pattern. In comparison, the implementation from [28] took 124 s.

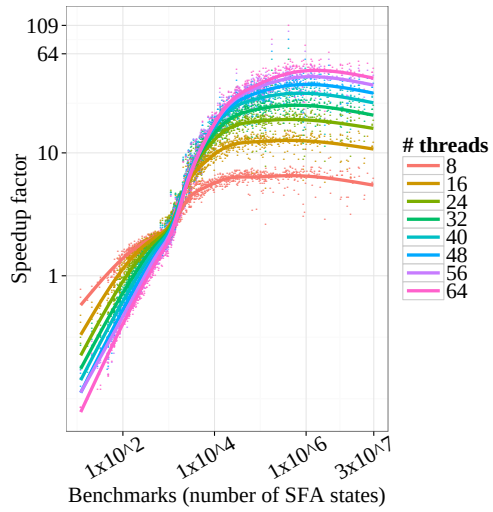
B. Parallelization

Fig. 5 describes the speedup of our parallel SFA construction algorithm with fingerprints, hashing and parameterized transposition over our fastest sequential version (which uses hashing and parameterized transposition). Thus, the depicted speedups are solely from parallelization. Axes use a logarithmic scale. Thus, the overall speedup compared to the baseline sequential algorithm can be calculated by multiplying the parallel speedup and the best optimized sequential speedup.

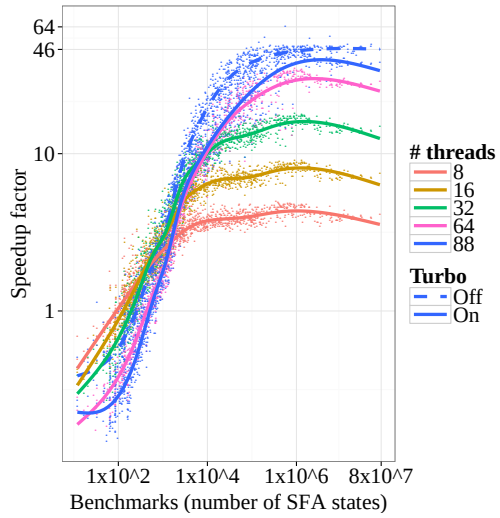
The maximum speedups obtained from parallelization on the AMD and the Intel systems are 108.9x with 64 threads and 46.1x with 88 threads, with median speedups of 4.9x and 4.6x, respectively. The reason why median speedups of PROSITE benchmarks are different from their maximum speedups is that there is not enough work in the case of small benchmarks which generate a smaller numbers of SFA states. In this situation, a smaller number of threads tends to show better performance, because more threads incur a penalty from cache coherence transfers. (Larger SFA sizes do not fit into the CPU caches anymore and rely more on data from main memory.) With small benchmarks, CAS operations employed for work-stealing become more competitive because owner and thief threads try to steal the same state at the same time. Except small benchmarks, speedups are proportional to the number of threads and it is impossible to obtain those speedups without thread-local queues.

The Intel system allows to dynamically disable the CPUs' turbo-boost feature. For 88 threads, we obtained the speedups without turbo-boost ("Turbo Off" in Figure 5b). Because the sequential baseline benefits more from turbo-boost relative to the parallel version, the obtained speedups from parallelization are slightly higher if turbo-boost is disabled. Note that enabled turbo-boost resulted in the faster *absolute* execution times.

To illustrate the benefits of our thread-local queues, we compared them to an implementation based on a concurrent queue from the Intel Threading Building Blocks (TBB). The r500 pattern with 8, 16, 32, 64 and 88 threads with our queues resulted in execution times of 1.43 s, 0.76 s, 0.38 s, 0.21 s and 0.16 s, whereas the TBB concurrent queue version took 1.44 s, 0.77 s, 0.39 s, 0.89 s and 1.00 s. To investigate the difference between them, we employed the Linux perf C2C (cache to cache) tool [29], which we



(a) on the AMD system



(b) on the Intel system

Figure 5: Speedup of the parallel implementation over the best sequential implementation (i.e., fingerprints + hashing + parameterized transposition) for 1250 PROSITE protein patterns [15]. Parallelization is profitable already for small SFAs of $\geq$ 100 SFA states. For four outliers, the AMD platform exhibits a super-linear speedup.

used to obtain the number of loads that hit in a modified cacheline (HITM). HITM loads are an evidence of data communication across cores resulting from either true or false sharing of data within a cache-line. For the r500 pattern on the Intel system at a threshold of $\geq$30 CPU cycles, the total number of HITM loads obtained for our thread-local queues was 2630, versus 5637 for the version using a TBB queue. The most frequently detected HITM cachelines with the TBB queue were related to the atomic operations on the TBB queue's internal state, which we suspect to cause higher synchronization overhead and eventually the scalability problem of the TBB queue. In particular, the access offsets within the TBB queue's HITM cachelines were constant, indicating *true* sharing. Compared to our thread-local queues, which are single-producer, single-consumer in the absence of theft from work-stealing, and single-producer, multiple-consumer if work-stealing occurs, the TBB queue is a multiple-producer, multiple-consumer queue, which increases contention and hence the HITM rate.

C. Compression

Table II illustrates the experimental results of six benchmarks on the Intel system, four of them are intractable without compression. The two tractable benchmarks have been added to show the run-time overhead of compression. For the tractable benchmarks, we set our memory manager's threshold to 200 GB to force compression which would otherwise be unnecessary on the 512 GB test platform. "n/a" denotes benchmarks which were intractable without compression. (The theoretical memory requirements of the intractable benchmarks where computed from a benchmark's number of SFA states, as uncompressed SFA states have constant size.) The compression ratios in the last column are $\geq$ 17 because of structural properties of FAs stated in Section III-C. Because compression mitigates the space constraints, bigger problem sizes become tractable. As shown with the tractable benchmarks, compression incurs a serious performance overhead, which is only justified for otherwise intractable benchmarks. Thus, our three-phase algorithm initiates compression only if an SFA would not fit into memory otherwise (rather than compressing already from the beginning).

D. SFA Matching

With the r500 pattern, we contrast our parallel SFA construction and parallel SFA matching scheme with sequential string pattern matching on the Intel system. Since the execution time of the sequential matcher is proportional to the length of the input string, we measured the absolute execution time (7.94 s) of the sequential matcher with the input string of size 1 GB. From this the execution times of the sequential matcher were estimated for different input sizes. The execution time of the parallel matching is

Table II: Performance and Size Comparison with and w/o Compression.

Number of States		w/o compr.		with compr.		Compr.
DFA	SFA	Size (MB)	Time (s)	Size (MB)	Time (s)	Ratio
2,557	74,624,878	381,632	42	12,622	1,015	30
2,980	40,956,096	244,098	28	13,172	436	19
6,132	17,795,082	436,478	n/a	12,153	824	18
6,419	20,559,280	527,880	n/a	15,495	1,212	17
6,549	47,076,417	1,233,214	n/a	29,610	2,700	21
7,025	23,975,400	673,709	n/a	20,106	1,641	17

linear in the number of threads, because the input string is partitioned among threads evenly. Because our parallel SFA construction with 88 threads resulted in 0.16 s, 20 MB is the break-even input size for this benchmark when 88 threads do string pattern matching in parallel. Thus, even for comparatively small input sizes it already pays off to construct an SFA and match in parallel, rather than to use a sequential matcher.

V. Related Work

To the best of our knowledge, there are no competing approaches for parallelizing SFA construction. Thus, we compare our method to research efforts that parallelize FA matching itself.

Locating a string in a larger text has applications with text editing, compiler front-ends and web browsers, internet search engines, computer security, and DNA sequence analysis. Early string searching algorithms such as Aho–Corasick [30], Boyer–Moore [31] and Rabin–Karp [32] efficiently match a finite set of input strings against an input text.

Regular expressions allow to specify potentially infinite sets of input strings. Virus signature specifications in intrusion prevention systems [33]–[35] and DNA sequences [36], [37] are common applications of regular expression matching with DFAs.

Parallel algorithms for DFA membership tests have already stirred considerable research interest. Ladner et al. [2] employed parallel prefix computations for DFA membership tests with Mealy machines. Hillis and Steele [3] employed parallel prefix computations on the 65,536 processor Connection Machine. A survey by Ravikumar [38] shows how DFA membership tests can be expressed as a chained product of matrices. Because of the underlying parallel prefix computation, all three approaches perform a DFA membership test on input size n in $\mathcal{O}(\log(n))$ steps, but they require n processors.

Recent work focused on speculation to parallelize DFA membership tests. Holub and Štekr [6] were the first to partition the input string into chunks and match them in parallel. Their speculation introduces redundant computation, which constrains the obtainable speedup for general DFAs to $\mathcal{O}(\frac{|P|}{|Q|})$, where $|P|$ is the number of processors, and $|Q|$ is the number of DFA states. Their algorithm degenerates to a speed-down when $|q|$ exceeds the number of processors.

Jones et al. [7] found that the IE 8 and Firefox web browsers spend 3–40% of the overall execution-time on parsing HTML documents. Jones et al. employ speculation to parallelize token detection (lexing) of HTML parsers. Similar to the k-local automata of Holub and Štekr, they use the preceding k characters of a chunk to synchronize a DFA to a particular state. Unlike k-locality, which is a static DFA property, Jones et al. speculate the DFA to be in a particular, frequently occurring DFA state at the beginning of a chunk. Speculation fails if the DFA turns out to be in a different state, which requires re-matching of the respective chunk. Lexing HTML documents results in frequent matches, and the structure of regular expressions is reported to be simpler than, e.g., virus signatures [10]. Speculation is facilitated by the fact that the state at the beginning of a token is always the same, regardless where lexing started.

The speculative parallel pattern matching (SPPM) approach by Luchaup et al. [8], [10] uses speculation to match the increasing network line-speeds faced by intrusion prevention systems. SPPM DFAs represent virus signatures. Like Jones et al., DFAs are speculated to be in a particular, frequently occurring DFA state at the beginning of a chunk. SPPM starts the speculative matching at the beginning of each chunk. With every input character, a speculative matching process stores the encountered DFA state for subsequent reference. Speculation fails if the DFA turns out to be in a different state at the beginning of a speculatively matched chunk. In this case, re-matching continues until the DFA synchronizes with the saved history state (in the worst case, the whole chunk needs to be re-matched). A single-threaded SPPM version is proposed to improve performance by issuing multiple independent memory accesses in parallel.

Ko et al. [11] devised a speculative parallel DFA membership test for multicore, SIMD, and cloud computing environments. Their SIMD version employs the `gather` instruction from the x86 AVX2 instruction-set extension. The proposed membership test uses structural properties of the underlying DFA to partition the input string and speed up matching. Unlike the previous approaches, their proposed speculation technique is *failure-free*, which means that speed-downs are avoided altogether.

Mytkowicz et al. [12] provide a parallel matching algorithm for DFAs, which breaks the dependencies of state transitions by simulating transitions from all possible start states on each input character. With their enumeration algorithm, the number of simulated transitions decreases over time if two simulations converge to the same successor state.

To remove the overhead from speculative parallel pattern matching, SFAs have been introduced by Sin'ya et al. [13]. In our work, we optimize their SFA construction algorithm.

Stern et al. [39] applied hashing to save memory because the verification of complex protocols suffered from the state explosion problem. Although we have the same state explosion problem, we approached it in a non-probabilistic way, whereas they considered a probabilistic method by compacting states in the hash-table.

VI. Conclusion

Although string pattern matching with FAs is widely used, parallelizing its sequential algorithm has been hampered because of the data-dependency between state transitions. SFAs have been introduced to simulate parallel execution of an FA. Although SFA matching achieved competitive

speedups, constructing SFAs for real-world problem sizes was prohibitive because of execution time and memory requirements due to the SFA state explosion problem.

In this paper, we presented several optimization methods for the SFA construction algorithm on multicore architectures. We adopted fingerprints and introduced hashing to reduce state comparisons and set membership tests to $\mathcal{O}(1)$ in most cases. Parameterized transposition of the transition table ensures cache locality of memory accesses. Our proposed algorithm is nonblocking and exploits sources of parallelism at various granularities. We proposed thread-local queues together with work-stealing for efficient, dynamic work distribution among threads. Our construction algorithm dynamically switches to in-memory compression of SFA states if the construction would not fit into memory otherwise. We have shown that data compression is effective with SFA states: our data compression scheme increases the computable problem sizes by up to a factor of 30x. Overall speedups over the sequential baseline are up to 312x on the AMD evaluation platform, and 193x on the Intel platform.

Acknowledgment

This research was supported by the Austrian Science Fund (FWF) project I 1035N23, and by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT & Future Planning under grant NRF2015M3C4A7065522.

References

[1] S. Yu, Q. Zhuang, and K. Salomaa, "The state complexities of some basic operations on regular languages," *Theor. Comput. Sci.*, vol. 125, no. 2, pp. 315–328, 1994.

[2] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[3] W. D. Hillis and G. L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, pp. 1170–1183, Dec. 1986.

[4] J. Misra, "Derivation of a parallel string matching algorithm," *Inf. Process. Lett.*, vol. 85, no. 5, pp. 255–260, Mar. 2003.

[5] D. P. Scarpazza, O. Villa, and F. Petrini, "Peak-performance DFA-based string matching on the Cell processor," in *21th International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[6] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," in *Proceedings of the 14th International Conference on Implementation and Application of Automata*, 2009, pp. 54–64.

[7] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík, "Parallelizing the web browser," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 7–7.

[8] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Multi-byte regular expression matching with speculation," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. Springer, 2009, pp. 284–303.

[9] X. Wang, K. He, and B. Liu, "Parallel architecture for high throughput DFA-based deep packet inspection," in *2010 IEEE International Conference on Communications*, 2010, pp. 1–5.

[10] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 438–451, 2011.

[11] Y. Ko, M. Jung, Y. Han, and B. Burgstaller, "A speculative parallel DFA membership test for multicore, SIMD and cloud computing environments," *International Journal of Parallel Programming*, vol. 42, no. 3, pp. 456–489, 2014.

[12] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines." Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2014.

[13] R. Sin'ya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *42nd International Conference on Parallel Processing (ICPP)*, Oct 2013, pp. 220–229.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Addison-Wesley, 2006.

[15] PROSITE protein database, version 20.114, "http://prosite.expasy.org," retrieved Jan. 2017.

[16] A. Gattiker, E. Gasteiger, and A. Bairoch, "ScanProsite: a reference implementation of a PROSITE scanning tool," *Applied Bioinformatics*, vol. 1, no. 2, 2002.

[17] M. O. Rabin, "Fingerprinting by random polynomials," Center of Research in Computer Technology, Harvard University, Tech. Rep. TR-15-81, 1981.

[18] A. Z. Broder, "Some applications of Rabin's fingerprinting method," in *Sequences II: Methods in Communications, Security, and Computer Science*. Springer, 1993, pp. 143–152.

[19] G. Pike and J. Alakuijala, "Cityhash, a family of hash functions for strings, https://github.com/google/cityhash," retrieved Oct. 2016.

[20] G. Pike, "Farmhash, a family of hash functions, https://github.com/google/farmhash," retrieved Oct. 2016.

[21] Intel Intrinsics Guide, "https://software.intel.com/sites/landingpage/IntrinsicsGuide," retrieved Oct. 2016.

[22] H. Choi and B. Burgstaller, "Non-blocking parallel subset construction on shared-memory multicore architectures," in *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*. Australian Computer Society, Inc., 2013, pp. 13–20.

[23] Squash compression benchmark GitHub page, "https://quixdb.github.io/squash/," retrieved Jan. 2017.

[24] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley, 1997.

[25] D. Raymond and D. Wood, "Grail: A C++ library for automata and expressions," *Journal of Symbolic Computation*, vol. 17, pp. 17–341, 1995.

[26] Grail+ project web site, "http://137.149.157.5/Mirrors/www.csd.uwo.ca/research/Grail," retrieved Oct. 2016.

[27] PAPI project web site, "http://icl.cs.utk.edu/papi/overview/index.html," retrieved Oct. 2015.

[28] R. Sinya, "Regen: regular expression generator, engine, JIT-compiler, http://sinya8282.github.com/Regen/," retrieved Mar. 2017.

[29] Perf C2C blog, "https://joemario.github.io/blog/2016/09/01/c2c-blog/," retrieved Jun. 2017.

[30] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[31] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[32] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

[33] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06. IEEE Computer Society, 2006, pp. 2–16.

[34] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. ACM, 2003, pp. 262–271.

[35] M. Roesch, "Snort—lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration*, ser. LISA '99. USENIX Association, 1999, pp. 229–238.

[36] C. Sigrist, L. Cerutti, E. De Castro, P. Langendijk-Genevaux, V. Bulliard, A. Bairoch, and N. Hulo, "PROSITE, a protein domain database for functional characterization and annotation," *Nucleic acids research*, vol. 38, p. D161, 2010.

[37] B. Boeckmann, A. Bairoch, R. Apweiler, M. Blatter, A. Estreicher, E. Gasteiger, M. Martin, K. Michoud, C. O'Donovan, I. Phan *et al.*, "The SWISS-PROT protein knowledgebase," *Nucleic acids research*, vol. 31, no. 1, p. 365, 2003.

[38] B. Ravikumar, "Parallel algorithms for finite automata problems," in *IPPS/SPDP Workshops*, vol. 1388. Springer, 1998.

[39] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *In Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer-Verlag, 1995, pp. 206–224.