Ant Colony-inspired Parallel Algorithm to Improve Cryptographic Pseudo Random Number Generators

Jörg Keller*, Gabriele Spenger* and Steffen Wendzel†
*Faculty of Mathematics and Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
Email: joerg.keller@fernuni-hagen.de, gabriele@spenger.org
†Faculty of Computer Science, Worms University of Applied Science, 67549 Worms, Germany
Email: wendzel@hs-worms.de

***Abstract*—We present a parallel algorithm to compute promising candidate states for modifying the state space of a pseudo-random number generator in order to increase its cycle length. This is important for generators in low-power devices where increase of state space is not an alternative. The runtime of the parallel algorithm is improved by an analogy to ant colony behavior: if two paths meet, the resulting path is followed at accelerated speed just as ants tend to reinforce paths that have been used by other ants. We evaluate our algorithm with simulations and demonstrate high parallel efficiency that makes the algorithm well-suited even for massively parallel systems like GPUs. Furthermore, the accelerated path variant of the algorithm achieves a runtime improvement of up to 4% over the straight-forward implementation.**

I. Introduction

Pseudo-random numbers are an important ingredient in a wide range of cryptographic protocols and applications, including applications in resource-constrained environments such as RFID chips or Internet of Things. There, the pseudo-random number generators (PRNGs) can only use little energy, i.e. use simple algorithms, but still must provide a decent level of security. One of the important criteria for a PRNG is the cycle length, i.e. the number of outputs until the sequence of outputs will repeat, but there are more criteria such as good statistical properties of the output sequence, forward and backward secrecy to name a few. Hence it is quite complicated to design a PRNG with a moderate state space size (because of resource constraints such as energy from a battery) that can provide these properties. For a PRNG that has already been investigated with respect to above criteria but where an increase in cycle length is desirable, we have proposed in previous work a method that only modifies a small number of state transitions to increase cycle length notably [1]. In order to find which state transitions to change, the state space of the PRNG has to be sampled, which is computationally intensive, and thus calls for the use of parallel computing.

In this work, we present a parallel algorithm to find promising states, called *candidate* states, for the modification of state transitions. The parallel algorithm is inspired by the behavior of ants, that tend to follow trails where other ants have already passed. We demonstrate the advantage of our algorithm over a straight-forward parallel implementation by simulation. While the asymptotic parallel efficiency of the parallel algorithm is highly dependent on the structure of the state transition graph, experiments indicate a good parallel efficiency (70%) in practice even for 1000 threads. Thus, together with its regular structure, the algorithm is suited for massively parallel computing engines like GPUs.

The remainder of this paper is organized as follows. In Section II we summarize background information on PRNGs. In Section III we present a parallel algorithm to identify promising candidate states for transition modification. In Section IV we demonstrate the suitability of our parallel algorithm by simulation experiments, and Section V gives conclusions and an outlook to future research.

II. Basics

Pseudo-random number generators (PRNGs) are used to generate (pseudo-)random numbers that are frequently used in communication protocols, be it as a nounce, a challenge, or for some other purpose. Between seedings, and while no additional entropy bits are input to the PRNG, on each call it outputs a value that depends on the current state, and transitions to a follow-up state by applying a state transition function on the current state. Thus, it works like a finite state automaton without input. A hash chain, i.e. a cryptographic hash function repeatedly applied to some initial value, is also used in cryptographic protocols, e.g. in Lamports authentication protocol [2]. After hashing the initial value once, the hash function works on the set of hash values much like the above transition function on the set of states. Other cryptographic primitives such as stream ciphers might be modelled in this manner as well.

PRNGs in resource-constrained systems such as mobile sensors typically use a state space of moderate size, because e.g. in 8-bit systems the increase of the state space by 8 bits leads to one more instruction for each addition or logical operation, which in turn increases the energy consumption per cryptographic operation, and thus puts a load on the battery. Hence we see hash functions with low computational load and 64-bit output like SipHash[1] or BLAKE2s[2], and PRNGs with state spaces of similar size, such as AKARI [3].

There are a number of general security requirements for cryptographic primitives like forward secrecy and backward

[1]https://131002.net/siphash/siphash.pdf
[2]see RFC 7693

DOI 10.1109/SPW.2017.31

secrecy [4] and PRNG-specific models such as [5] and [6], or suites that test the output of PRNGs for randomness such as the Marsaglia suite of Tests of Randomness [7] and the NIST test suite [8]. Still, if the cycle length of the primitive is short, patterns of output bits can be stored and repetition detected. A practical example for this is the attack on A5/1 (cf. [9], [10]). Thus, a long cycle length is a prerequisite for enabling forward secrecy.

A PRNG can be modelled as a deterministic state transition function $f : M \to M$ mapping a finite state space of size $n = |M|$ to itself. If a single state is interpreted as a node and the transition between a state and its unique successor state is interpreted as an edge, the result is a directed graph $G_f = (V; E)$ with $V := M$ and $E := \{(x, f(x)) \,|\, x \in M\}$, where each node has exactly one outgoing edge (deg-1 graph). The structure of the generated graph provides information about the behavior of the primitive. For non-bijective transition functions, the graph typically consists of several weakly connected components. Each of these components consists of one cycle and generally several trees with roots located on the cycle. The trees tend to be very ragged. Figure 1 depicts the structure of a component.

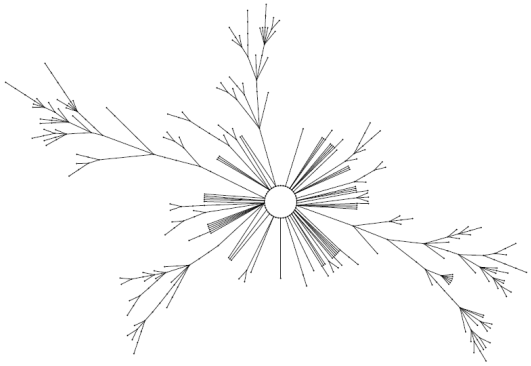

Fig. 1. Typical connected component of a state transition graph, taken from [11].

Properties of the graph include the number and sizes of the connected components, length of the cycles and maximum depth of the trees. In order to identify all connected components of a graph, the complete state space would have to be analyzed, e.g. by a depth first search (DFS). The cycles can be detected by starting from the unique back edge in each component. If the state space is too large for complete exploration, a part of the state space can be analyzed, accepting the fact that one or several components might be missed. Still, this approach can provide valuable information about the expected state space structure. The typical approach is to randomly select nodes as *starting points*, and to follow the path from each starting point until a cycle is reached. The number of followed paths, which influences analysis time, does not need to be very large: as the expected number of components is small (see below), already a small number of sample paths through the state graph will hit all of the larger components, and provide their cycle lengths. Please note: as the components are much larger than their cycles, a short cycle in such a component affects many seed states in the state graph.

In [12] an analysis method of the state space is presented that avoids the large memory requirements of depth-first search, where each node must be marked as visited, which is clearly impossible if $n \geq 2^{40}$. The idea is to only store certain nodes while traversing the tree. If only the nodes are stored that are reached after $2, 2^2, 2^3, \ldots$ steps taken since the start value (so called anchors), the required memory usage is logarithmically in the number of steps, i.e. $O(\log n)$, and thus very low. A cycle is reached if the newest anchor is reached again. Figure 2 illustrates the process of cycle detection. The low memory requirement comes with a time overhead: the algorithm might need twice as long as would be needed in the optimal case.

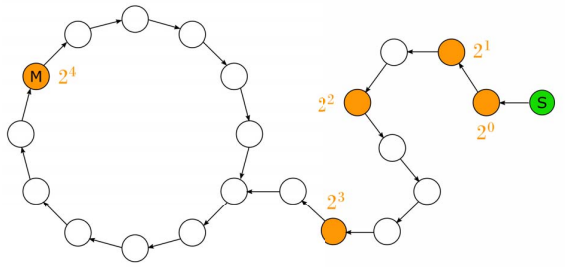

Fig. 2. Cycle detection.

This runtime can be improved by spending more memory, but less than 1 bit per node. One can store the nodes reached after $k, 2k, 3k, \ldots$ steps, and check in each step if one of the stored nodes is reached again. This reduces the overhead to at most k additional steps, but requires a search data structure of size $M = O(m/k)$ for a path of length m, which must be queried in each step. Hence, the query time must be constant (at least if amortized over many queries), for example by using a hash table with low utilization. The parameter k can be chosen given the path length m and the memory size M. While the latter is known for the computer to be used, the path length can only be guessed. For a randomly chosen state transition function, the expected path length is $O(\sqrt{n})$ [13]. However, there is no guarantee for this, so if the data structure runs out of memory, it must adapt dynamically by throwing away every other node stored, and increasing k to twice its former value. The data structure can be used to shorten the time for sampling further paths: if one keeps the nodes stored from previous paths, then one can stop following a further path if one of these stored nodes is reached. Because of the tree structures in the components (cf. Fig. 1) the paths meet sooner or later in the tree. At least, if two paths are in the same component, another complete walk around the cycle can be avoided for the second path. As both the expected tree path[3] and cycle lengths are $O(\sqrt{n})$ with comparable constant factors

[3]The tree path is the path from the start node until the entry into the cycle.

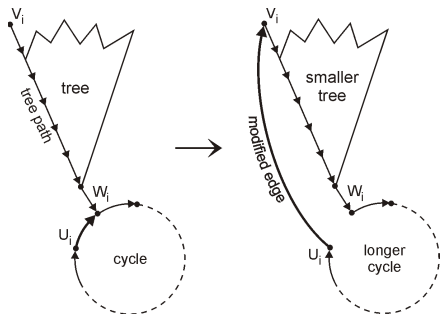

Fig. 3. Breaking up a cycle to increase cycle length.

[13], this on average should reduce the length by a factor of 2. As the number of weakly connected components is expected to be small ($0.5 \cdot \log n$, see [13]), many paths will be in the same components, and thus it normally pays off to increase k to be able to store nodes from all paths sampled so far.

The runtime of this algorithm is proportional to the average length of the paths and the number of starting points. As a small number of starting points suffices, the average path length can be around 2^{40} and still yield reasonable analysis time. This restricts n to 2^{80} if path lengths are around $\sqrt{n}$ (see above), which allows analysis of PRNGs or hash chains on a 64-bit state space. The resulting tree and cycle structure for these samples might provide valuable insights with respect to the security of the algorithm. Also, the sizes of the connected components can be guessed from the fractions of starting points belonging to each component, within a confidence interval. Please note that there are comparable approaches for bijective functions, notably Knuth's algorithm [14] with an expected runtime of $O(n \log n)$, which can be made linear in time by using 1 bit per node, or improved by using stored nodes as described above.

III. Algorithms

For a PRNG with a short cycle, our strategy to increase the cycle length is to cut the cycle at some node (which we will call a *special* state in the following) by modifying the transition function to divert to a node somewhere deep in the tree [1], as illustrated in Fig. 3. There, the outgoing edge (u_i, w_i) of cycle node u_i is modified, and tree node v_i becomes the new successor of u_i. The resulting cycle length will be the sum of the previous cycle length and the length of the tree path from v_i to w_i.

This can be done for each component of notable size, and even multiple times within each component to achieve a larger increase of the cycle length[4]. As the number of components is expected to be small, with only a few components of notable size (see previous section), the total number of modified edges will be small as well, and can be stored in a lookup table. The modified PRNG state transition function can then be implemented as given by Alg. 1.

[4]Also other modifications can be applied: if e.g. a component of notable size has a short cycle and the trees are rather flat, then instead of enlarging this cycle it might be more advantageous to cut the cycle and modify the edge towards a deep tree in another component with a longer cycle, thus making the former component a tree of the other component [1].

Algorithm 1 Modified state transition function.

Precondition: s is the current state of the PRNG, TRANS is the original state transition function

```
1: function MODTRANSITION(s)
2:     if s is special state then        ▷ access to lookup table
3:         s_next ← new successor of s from lookup table
4:     else
5:         s_next ← TRANS(s)
6:     return s_next
```

Please note that the time spent in the lookup table still increases the execution time of the transition function, which could increase the energy consumption. Hence, we split the test — which will fail most of the time, because there are only few special states — into two parts: a very fast test, that fails in most cases, and a followup-test, that does the exact check but is executed only seldomly. The first test uses a property that is easily testable, e.g. that some bits of the state representation have a certain bit pattern. We call these nodes *candidates*. The only restriction imposed by this test is that states where the transition function shall be altered must be candidates, which is however no serious restriction, cf. [1].

The difficult task is to find a small number of special states and new edges going out from these special states. This must be done such that cycle lengths are increased and other properties like statistic behaviour of output is not harmed. While this has to be done only once, i.e. is an offline task, it requires to sample the state graph (see previous section), i.e. it might require 2^{40} executions of the the state transition function if the average path length is 2^{32} and 2^8 starting points are chosen. To do this in a reasonable time calls for a parallel algorithm.

If we ignore the case of encountering a component without a candidate (that case can be handled by additionally using anchors, cf. Sect. II), then the selection of special states can be done using the construction of the candidate graph, i.e. the graph of all candidate nodes reachable from the chosen starting points. An edge in the candidate graph between two candidate nodes c_i and c_j represents the unique path in the state graph vom c_i to c_j, and is annotated with the length of this path. When the candidate graph has been computed, the cycle lengths and the tree depths can be computed by DFS, and the special states can be chosen by changing edges such that the increase in cycle length is maximized. This can be repeated until all candidates of a connected component of the candidate graph are on a cycle, or a maximum number of special transitions is achieved. For details about the determination of special states from the candidate graph, we refer to [1].

The computation of the candidate graph follows a simple paradigm: the paths originating from the starting points are followed, and if two paths meet, only one of them is followed

further. While we are following a path, we record all candidates that we visit. This leads to the following basic parallel algorithm, cf. Alg. 2.

Algorithm 2 Parallel algorithm to compute candidate graph.

Precondition: S is the set of starting points, $m = |S|$, ISCAND checks if a state is a candidate.

```
1: function COMPCGRAPH(S)
2:     for all i ← 1 to m do                          ▷ Parallel Loop
3:         p_i ← s_i ∈ S
4:         repeat
5:             repeat
6:                 p_i ← TRANS(p_i)
7:             until ISCAND(p_i)        ▷ Reached next candidate
8:             ADDEDGE(p_i)
9:         until p_i is already visited by other path
```

Each thread follows a path from one candidate to the next. Then it checks if that candidate has been already visited by another thread. If so, then the thread stops, and the other thread continues. If not, then the thread follows this path further in the next round. If two threads reach a candidate in the same round, then the one with the smaller ID continues. Please note that we hide some details here. First, to construct the candidate graph, not only nodes but also edges with distances must be added. Also, one thread will reach the cycle and there meet a candidate visited earlier by itself, which must also be detected. Finally, it is not guaranteed, that a path contains a further candidate (although this is unlikely), so that in addition, other measures are necessary to detect if a cycle has been reached (cf. Sect. II).

The parallel algorithm partitions each tree in the graph into *chains* that correspond to the paths that the threads follow. A chain always starts in a starting point s_i and ends in a candidate reached by more than one path and where s_i is not the closest starting point. Put otherwise, when the path from s_i reaches the candidate, it has been visited before by another thread. Figure 4 illustrates this with five starting points A to E, where the paths starting in A and B meet in candidate X, and only the path from A is followed further. Similarly, the paths starting in C, D and E meet in candidate Z, where only the path from C is followed further until candidate Y, where it meets the path from A, that is followed further around the cycle. The different chains are indicated by different colors.

The runtime of the algorithm is then determined by the longest chain, i.e. by the longest sequence of candidates found from this set of starting points, multiplied with the average distance between candidates. The average distance between candidates is n/c if there are n possible states and c candidate states, assuming that the transition function is random enough that the deterministic choice of the candidates makes their distribution in the graph similar to a random distribution. The standard deviation is high, however, so that the maximum distance occurring in one round can be much higher than the average, leading to load imbalance and idling threads. In order

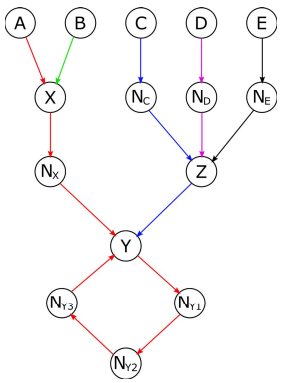

Fig. 4. Parallel sampling of paths.

to avoid this, each thread follows several paths in each round, so that the resulting runtime better approaches the average. In addition, this occurs quite naturally as the number of followed paths m can be larger than the number p of threads available, even considering a massively parallel environment like a GPU with several thousand hardware threads. The technique to follow multiple paths per round with subsequent query for visited candidates has been used before in a parallel program [15], although with a different intention: by bundling multiple queries and querying more seldomly, the high communication cost in a message-passing maching could be amortized. This does not play a role in our current research as the graph is small enough to be kept in a shared memory. Still, in a GPU the access to the global memory is slow, so that infrequent coordinated access helps performance.

If the number of followed paths gets smaller after some time, a load balancing can be performed to maintain load balance as far as possible. As soon as the number of followed paths gets smaller than the number of threads, a load imbalance necessarily occurs. This load imbalance hurts if the difference between the maximum chain length and the majority of chain lengths is large. As the small example in Fig. 4 illustrates, after two rounds only two of the five chains are left, and after three rounds only one chain is left, which continues for another four rounds.

The runtime could be improved if such long chains could progress faster relative to other paths. Note that this is possible as a thread follows several paths in each round, so that instead of advancing t paths to the next candidate, the thread could advance $t-2$ paths to the next candidate and one path to the next but one candidate. Thus, the latter path would progress twice as fast as the other paths. As the thread now only handles $t-1$ instead of t paths, a different distribution of paths onto threads is necessary, but presents no problem. Unfortunately, the long chains are only known at the end of the algorithm, so it is not clear which path should progress faster than the others.

In order to still improve load balance, we borrow an analogy from nature: when an ant meets the path of other ants, it tends

to follow this path, thus strengthening this path by placing further pheromone. Ant colony algorithms have successfully been used to solve problems related to graph theory, e.g. in [16]. Here, if a path meets a candidate that has been already visited by another path, it strengthens that path by "donating" its own time slot to the other path, which is possible as the first path need not be followed further. Hence, if the other path is a long chain, it will progress faster. While this simple heuristic will not help a long chain where no other paths end, that situation is unlikely due to the ragged structure of the trees in random deg1-graphs. Obviously, this donation cannot be continued long in a linear fashion, because each thread follows at most m/p paths in one round, and therefore an acceleration of a path that got donations from more than m/p others would slow down a round. Hence, the most advantageous form of acceleration must be found out by experiments, as the optimum acceleration cannot be determined at runtime. Furthermore, the donation can be transitive, i.e. a path that got donations from other paths, and donates its own time to another path, would also donate the time it got itself from others. The experiments in Section IV will illustrate that simple strategies are sufficient to turn this nature-inspired analogy into a real advantage.

IV. Experiments

To quickly assess the advantage of our parallel algorithm with path acceleration over the straight-forward parallel implementation, we use a simulator. The simulator reads in a candidate graph that has been produced in our previous research [1]. It then simulates the threads one by one and round by round. As the candidate graph structure is already available, the inner repeat-until loop from Alg. 2, that searches for the next candidate, can be reduced to one step. Still, as the distance between succeeding candidate nodes is stored in the graph, the exact runtime of a round (in terms of maximum number of calls to function `trans` per thread) can be given.

The simulator is applied to a graph with paths from $m = 10^4$ randomly chosen starting nodes, using as transition function the cryptographic hash function MD5 with output restricted to 64 bits[5] The candidate set was defined as the set of nodes with bits 4 to 25 set to 1.

The simulator is run in several configurations: either with $p = 100$, 500 or 1000 threads, to test differing ratios of m/p. We use two simple donation functions: either linear or logarithmic in the number of donated time slots (up to m/p). Additionally, we use the straight-forward implementation, where no donation occurs. For comparison we also give the sequential runtime. Please note that by runtime, we mean the number of edges from one candidate to the next that a thread follows during the algorithm. We skipped the more exact measure of calls to function `trans` in order not to model the load balancing.

Table I presents the results for the different configurations. We see that for $p = 100$, the linear donation strategy brings a runtime advantage of 4%, which seems small but is notable given that less than 100 rounds are done in the parallel algorithm, so that it will take a while before time slot donation can show effect. Also, for an algorithm with a sequential runtime of many hours this increase still saves a minute in the parallel version. For larger p, the advantage is smaller, as more edges have already been processed before the donation can show effect. The logarithmic donation strategy brings a small advantage for $p = 100$ but is slightly slower than the straight-forward implementation without time slot donation. Hence, the linear donation strategy should be chosen. We also note that the parallel efficiency of our algorithm is high: still 70% for $p = 1000$, demonstrating scalability for massively parallel computing engines like GPUs. While it would be desirable to formulate parallel efficiency as a function of p, this is not possible as it also depends on the structure of the state graph.

TABLE I
Simulated runtimes of parallel algorithm for different thread count and donation strategies.

	donation strategy		
p	no	linear	logarithmic
1	155,524		
100	1,614	1,553	1,598
500	379	371	380
1000	226	225	229

We also tested the algorithm with the logistic map $f(x) = a \cdot x \cdot (1 - x)$ for $a = 3.99$ and x implemented by double precision IEEE754-compliant arithmetic, as an example of a chaotic PRNG [1], but that graph is too small and too flat. It comprises only 20,295 edges outside cycles for 10,000 starting points. Thus, no runtime difference between the different donation strategies could be observed. However, the parallel efficiency of the algorithm is high: 99% for $p = 100$ to 88% for $p = 1000$.

V. Conclusion

We have presented a parallel algorithm for finding promising candidate states to modify the state transition function of a pseudo random number generator. Use of these candidates allows to increase the cycle length notably, which is helpful if the state space itself cannot be enlarged due to resource constraints such as performance and energy in embedded devices. The inherent load balancing problems of this algorithm can be resolved by the use of an ant colony-like strategy: paths that are not followed further because of meeting another path donate their time to the other path that can then progress faster. This illustrates once more how nature can inspire improvements in security-related algorithms. The resulting parallel algorithm exhibits regular structure and high efficiency even for large thread count, and thus is suited for massively parallel computing engines like GPUs.

Our future work will include further experiments to fine-tune the "donation" of time-slots in order to maximize performance, and to transfer our simulation to a real implementation on a GPU. The performance possible on this massively parallel

[5]We are aware that MD5 is outdated, and used SHA-3 in [1]. Both graph structures are quite similar to random deg-1 graphs, and thus should behave similarly with respect to the parallel algorithm.

processing device will also enable to tackle larger state spaces. We would also like to extend our work towards similar primitives such as stream ciphers. Here Spritz [17] might be a good candidate, as it would also allow to extend our work from non-bijective towards bijective transition functions.

REFERENCES

[1] G. Spenger and J. Keller, "Tweaking cryptographic primitives with moderate state space by direct manipulation," in *Proc. IEEE International Conference on Communications (ICC'17)*. IEEE, May 2017.

[2] L. Lamport, "Password authentication with insecure communication," *Commun. ACM*, vol. 24, no. 11, pp. 770–772, Nov. 1981.

[3] H. Martin, E. S. Millan, L. Entrena, P. P. Lopez, and J. C. H. Castro, "Akari-x: A pseudorandom number generator for secure lightweight systems," *11th IEEE International On-Line Testing Symposium*, vol. 0, pp. 228–233, 2011.

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[5] A. Desai, A. Hevia, and Y. L. Yin, "A practice-oriented treatment of pseudorandom number generators," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2002, pp. 368–383.

[6] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergniaud, and D. Wichs, "Security analysis of pseudo-random number generators with input:/dev/random is not robust," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 647–658.

[7] G. Marsaglia, "The marsaglia random number cdrom including the diehard battery of tests of randomness," 1995. [Online]. Available: http://www.stat.fsu.edu/pub/diehard/

[8] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "Statistical test suite for random and pseudorandom number generators for cryptographic applications: Special publication 800-22, revision 1a," 2010. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf

[9] A. Biryukov, A. Shamir, and D. Wagner, "Real time cryptanalysis of A5/1 on a PC," in *Fast Software Encryption*. Springer, 2001, pp. 37–44.

[10] J. Golic, "Cryptanalysis of alleged A5 stream cipher," in *Advances in Cryptology (EUROCRYPT '97)*, ser. Lecture Notes in Computer Science, W. Fumy, Ed. Springer Berlin Heidelberg, 1997, vol. 1233, pp. 239–255. [Online]. Available: http://dx.doi.org/10.1007/3-540-69053-0_17

[11] A. Beckmann, J. Fedorowicz, J. Keller, and U. Meyer, "A structural analysis of the A5/1 state transition graph," in *Proc. First Workshop on GRAPH Inspection and Traversal Engineering*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 99. Open Publishing Association, 2012, pp. 5–19.

[12] J. Keller, "Parallel exploration of the structure of random functions," in *Proceedings of the 6th Workshop on Parallel Systems and Algorithms (PASA) in conjunction with the International Conference on Architecture of Computing Systems, ARCS*. VDE, 2002, pp. 233–236.

[13] P. Flajolet and A. M. Odlyzko, "Random mapping statistics," in *Advances in Cryptology*. Springer, 1990, pp. 329–354.

[14] D. E. Knuth, "Mathematical analysis of algorithms," in *Proc. of IFIP Congress 1971, Information Processing 71*. North-Holland Publ. Co., 1972, pp. 19–27.

[15] J. Heichler, J. Keller, and J. F. Sibeyn, "Parallel storage allocation for intermediate results during exploration of random mappings," in *Proc. 20th Workshop Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS '05)*. GI, Jun. 2005, pp. 126–134.

[16] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on evolutionary computation*, vol. 1, no. 1, pp. 53–66, 1997.

[17] R. L. Rivest and J. C. N. Schuldt, "Spritz—a spongy RC4-like stream cipher and hash function." Cryptology ePrint Archive, Report 2016/856, 2016, http://eprint.iacr.org/2016/856.