Efficient Sampling of the Structure of Crypto Generators' State Transition Graphs

Jörg Keller

FernUniversität, LG Parallelität und VLSI, 58084 Hagen, Germany
Joerg.Keller@fernuni-hagen.de

Abstract. Cryptographic generators, e.g. stream cipher generators like the A5/1 used in GSM networks or pseudo-random number generators, are widely used in cryptographic network protocols. Basically, they are finite state machines with deterministic transition functions. Their state transition graphs typically cannot be analyzed analytically, nor can they be explored completely because of their size which typically is at least $n = 2^{64}$. Yet, their structure, i.e. number and sizes of weakly connected components, is of interest because a structure deviating significantly from expected values for random graphs may form a distinguishing attack that indicates a weakness or backdoor. By sampling, one randomly chooses k nodes, derives their distribution onto connected components by graph exploration, and extrapolates these results to the complete graph. In known algorithms, the computational cost to determine the component for one randomly chosen node is up to $O(\sqrt{n})$, which severely restricts the sample size k. We present an algorithm where the computational cost to find the connected component for one randomly chosen node is $O(1)$, so that a much larger sample size k can be analyzed in a given time. We report on the performance of a prototype implementation, and about preliminary analysis for several generators.

1 Introduction

Stream cipher generators, like the A5/1 in cellular telephones [1, 2] and pseudo-random number generators, are widely used in cryptographic communication protocols. Basically, they are finite state machines that are initialized into a state and then assume a sequence of states completely determined by their transition function $f : N \to N$, where N is their state space, i.e. a set $N = \{0, \ldots, n-1\}$. For given N and f, one can define the state transition graph $G_f = (V = N, E = \{(x, f(x)) : x \in N\}$. Such a directed graph, where each node has exactly one outgoing edge, has a number of weakly connected components (WCC), each consisting of one cycle and a number of trees directed towards their roots, where the roots sit on the cycle. One is interested in the number of the WCCs, their sizes and cycle lengths. The cycle length represents the generator's period, the component size the fraction of nodes that, when chosen as initial state, lead to a certain period. If the period length is too small, then this may hint towards predictability. Furthermore, as a cipher generator shall, in some sense,

randomize, its graph should look randomly as well. If its structure deviates significantly from expected values for random graphs with outdegree 1, this may hint towards a weakness or a backdoor. In this sense, the computation of such a graph's structure can be considered as a distinguishing attack.

Unfortunately, the period lengths of such generators cannot be derived analytically. Also, typical graph algorithms with techniques like pointer doubling fail for two reasons. First, the graph is typically of size $n = 2^{64}$ and more, and thus cannot be constructed in memory. Second, even if it could be constructed, the graph could not be explored completely as an algorithm with at least linear complexity may take longer than our lifetime.

Parallel algorithms have been devised [4] that can explore such a graph completely, even if it cannot be constructed in memory. Yet, if the graph's size renders a complete exploration infeasible, they can also be used to "scan" such a graph by sampling. One randomly chooses $k < n$ nodes, determines the WCCs they belong to (and detects the cycles in those components), and then extrapolates this result to the complete graph. As determining the WCC of one node may already need time $O(\sqrt{n})$, this restricts the sample size and thus the validity of the extrapolation. Our contribution is an algorithm that reduces this overhead to $O(1)$, and hence allows a much larger sample of nodes to be visited.

The remainder of this article is organized as follows. In Section 2 we briefly review the relevant facts and the previous work. In Section 3 we present the new algorithm. Section 4 reports on preliminary performance results and on findings for some generators. Section 5 concludes.

2 Relevant Facts and Previous Work

State transition graphs as defined in the introduction are called *mappings* in the literature. An example graph for $n = 16$ can be seen in Figure 1. It consists of two WCCs of sizes 12 and 4 with cycle lengths of 5 and 3. For functions f randomly chosen from the set of all functions from N onto itself, Flajolet and Odlyzko [5] have derived expected values for the size of the largest component (about $0.76n$), of the largest tree (about $0.5n$), the expected path length from any node to a cycle (about $\sqrt{n}$), the expected cycle length and the expected length of the longest cycle (both $O(\sqrt{n})$, with slightly different constants close to 1).

If one starts at a node x (called *starting node*), the only thing than can be done to find out which WCC it belongs to, is to follow the unique path starting in x, by repeatedly computing $x := f(x)$, until a cycle is reached. As there is only one cycle per WCC, and there is a unique node with smallest number on the cycle (called *cycle leader*), the number of that node uniquely characterizes the component. One can find out to be on a cycle by storing after a number of steps which node has just been reached (*marker node*), and checking after each step, whether any of the stored marker nodes has been reached again. If the distances between marker nodes are always doubled, $O(\log n)$ marker nodes suffice and the effort is only increased by a constant factor [3]. As the average path length

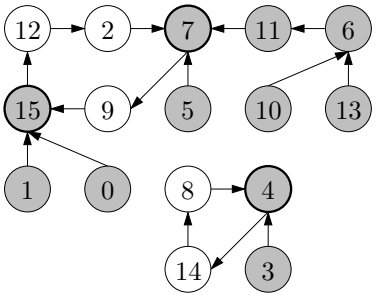

Fig. 1. An example graph.

and cycle length are both $O(\sqrt{n})$, a complete exploration has expected time $O(n\sqrt{n})$. This algorithm can be parallelized trivially, but even then the runtime is prohibitive. One can improve the runtime by keeping a store, as large as the main memory of all processors, for nodes called *pebbles* from which one already knows to which WCC they belong. Then if one reaches a pebble on a path, one can stop there.

We define a subset of the nodes called the *candidate* set. Only candidates can become pebbles. The candidate set is normally chosen independently of the function f, and in a manner that the membership to the candidate set can be computed efficiently from the node number, e.g. by requiring that some bits must be zero.

Several relationships between candidates and pebbles are possible. If every candidate indeed is a pebble, and if all pebble information is gathered in advance, then we have a completely static situation. Then in every step, we only have to check whether a candidate is reached, which can be done efficiently, and if so, we already know that we have reached a pebble. However, as we have to expect that a fraction of $1/e$ of the nodes are leaves [5], and as the candidate set is defined independently of the function f, we have to expect that $1/e$ of the pebbles are leaves as well. Pebbles on leaves are not worthwhile because only a single path can reach that pebble: the path originating in the leaf itself. If the candidate set is the set of all nodes, and if the pebbles are set during the exploration of the graph, then we have a completely dynamic situation. In this case, only a small fraction of the candidates can indeed become pebbles, because otherwise we would need $\Omega(n)$ memory resources to store the pebble information. While this scenario allows to place pebbles in a manner that takes into account the particular characteristics of the function f, it has a certain disadvantage. When following a path from some starting node, one has to check whether a pebble is reached after each step. Checking whether a candidate is a pebble requires a query to a data structure such as a search tree or a hash table and thus takes some time.

Therefore, we decided on a compromise [4]. Not every node can become a pebble. The size of the candidate set is only a fraction of n, typically $1/2^c$ as we

check whether the c lowermost bits of a node number are zero. The pebbles are placed while the graph is explored. To take the function f into account, we place pebbles in regular distances along the paths that we follow. This ensures that pebbles are spread out over the trees. The pebble data structure can be updated regularly to remove pebbles without visits in a certain time frame (similar to the LRU strategy in caches), which also gives room to place further pebbles. The pebble data structure can even be distributed over all processors to allow a pebble set that scales with machine size when we use a parallel cluster computer to follow many paths simultaneously [6].

However, the current approach suffers from a weakness. If n is really large, we cannot afford to explore the graph completely, because the effort to do so is $O(n \cdot l)$ if l is the average path length to a pebble[1]. What one can do is to restrict to a sample of $k < n$ starting nodes, chosen randomly among all nodes. If k_i of these k starting nodes belong to WCC i, then with standard techniques one can compute a confidence interval $[n_i - \delta_i : n_i + \delta]$ around $n_i = n \cdot k_i/k$ such that the size of WCC i lies in this interval with probability p. The effort for this sampling is still $O(k \cdot l')$, where l' is the average path length for the first k starting nodes. Normally, $l' > l$, as the pebbles could not yet be placed as well as after certain update improvements.

3 Efficient Sampling

We want to improve the algorithm of the previous section by taking into account the following observation: while for each starting node of the sample, only this node is attributed to a WCC, one has visited many nodes of this WCC! The bad thing is, that it is not clear how many of those nodes we have already visited in previous runs. If a path from a starting node reaches a pebble, we do not know how many of those nodes are on a path that has been visited in the past. Hence we extend our pebble data structure in order to be able to find this out.

Each pebble a now contains links all pebbles, from which paths reach a. Those pebbles are called *child pebbles*. Furthermore, we require that the tree root must be a pebble (so that we can guarantee that each path in the tree reaches a pebble), and also that each previous starting node is a pebble[2], if it contributed any newly visited starting nodes. In the unlikely event that a new starting node lies on a path visited before, it will not contribute any newly visited node and hence need not be considered further. If those requirements are maintained with every following starting node, then we can formulate the following invariant: *The pebbles, and the nodes on the paths between them, contain exactly the set of nodes already visited.*

[1] In the overwhelming majority of cases, a path ends in a pebble. Only in a tiny fraction of cases, a cycle is reached.

[2] This requires that only nodes that are candidates are chosen as starting nodes. This is however not a serious restriction as the set of candidates will be much larger than the set of starting nodes.

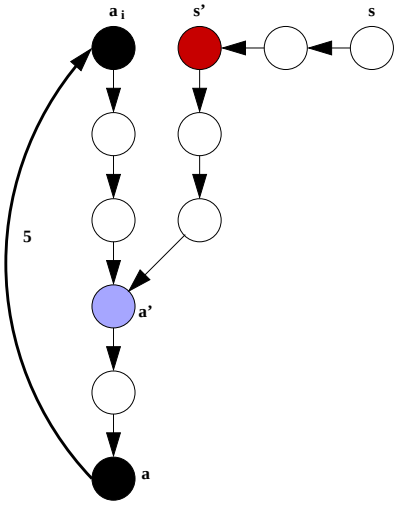

Fig. 2. Reaching a pebble.

Now, if the path from a new starting node s reaches a pebble a, we have only to find out where it met a path already visited. To do this, we visit the child pebbles of a, named a_1, a_2,..., which are in distances $d_1, d_2, \ldots$ from a. We will assume that the distances are in decreasing order. Now we find a node s' on the path from s to a in distance d_1 to a. If the distance from s to a is d (we assume $d > d_1$), then we can follow the path from s for $d - d_1$ steps. However, as we store marker nodes on the way, the effort normally is much smaller. Now we follow the paths from s' and a_1 step by step until they both reach the same node a'_1, where the paths meet. We do the same for all other child pebbles. If d' is the maximum distance of any a'_i from a, then the path from s to a contributes $d - d'$ newly visited nodes. If s is now made a pebble, and a further child pebble to a with distance d, then the invariant is maintained.

Figure 2 depicts an example situation. When starting from node s, the pebble a is reached after 7 steps. Pebble a has a pebble child a_i in distance 5. To find a node s' on the path from s to a with distance 5 to a, one starts at node s and follows the path for $2 = 7 - 5$ steps, to reach node s'. Now the paths from a_i to a and s' to a are followed simultaneously step by step, until both paths reach the same node a' after 3 steps. Hence, on the path from s to a, $5 = 2 + 3$ nodes have been visited for the first time. The node s will be made another child pebble of a with distance 7.

The overhead, defined as the number of evaluations of function f for computing d' is $O(d)$, if the number of child pebbles to a pebble is not more than a constant. This however can be achieved by adapting the pebble data structure (introducing new pebbles at places where paths from child pebbles meet) without changing the invariant. If we assume that the path length d from the

starting point to the pebble is not more than a constant factor longer than the number $d-d'$ of the newly visited nodes, the overhead will also be $O(d-d')$, and hence the overhead per newly visited node will be $O(1)$. The latter assumption is based on the fact that only a small fraction of the nodes will be pebbles and already visited nodes, and hence from a randomly chosen node, one will have on average have a long way to go until a path between pebbles is met. An additional overhead has to be accounted for the case where a starting node lies on a path that is already visited, and where no newly visited nodes will be contributed. However, if one assumes that the size n of the state space is so large that less than $10^{-3}n$ nodes can be visited in total, then the probability for this event is smaller than 10^{-3} and hence this event is seldom.

One may also argue that if a WCC i has a size n_i, then a starting node was chosen from this WCC with probability n_i/n, and thus the nodes attributed to WCC i were done so independently in the original algorithm. If the new algorithm chooses a starting node s from WCC i, then it attributes $d-d'$ nodes to that WCC, the newly visited nodes on the path starting in s. Yet, the average path length will depend on the placement of the pebbles, which will not directly depend on the WCCs, and so will the path length. Furthermore, if WCC i has had more visited nodes (in relation to its size) than other WCCs, then one has to take into account that the probability to choose an unvisited node from WCC i will sink below n_i/n, and the other WCCs will receive accordingly more starting nodes and thus more visited nodes, so that the balance is approached again.

4 Experimental Results

We have programmed a simple, sequential version of the new algorithm. As only values of n up to 10^7 are used for evaluation purposes, we could use a variant where a bit could be stored for each node, to find out whether this node was visited before. Thus, the algorithm immediately knows how many nodes have newly been visited on this path. The pebbles are placed randomly for the sake of simplicity, which will lead to a constant average distance between pebbles on a path. As overhead, we only counted the way from the first visited node to the next pebble. If we assume that on average, a path from a starting node will meet a known path in the middle between two pebbles, and that a pebble on average has two child pebbles, then we would have to increase the overhead by a factor of 6, because the length would double, and three paths would have been followed (two starting in child pebbles, one starting on the new path).

4.1 Performance Results

We tested our algorithm on a number of functions generated randomly with the help of the `lrand48` pseudo random number generator, with different seeds. We first tested functions of size $n = 10^6$. Figure 3 depicts the average number of nodes newly visited, and the corresponding average overhead, for a sequence of starting nodes. The x-axis represents the starting nodes as percentage of n, i.e.

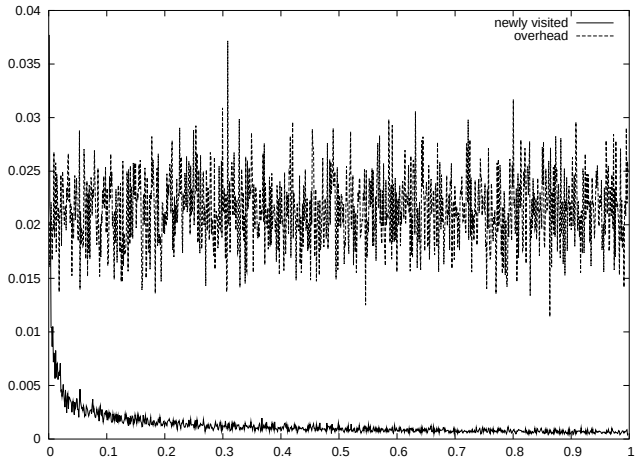

Fig. 3. Newly visited nodes and overhead per starting node for $n = 10^6$, in percentages of n.

1% of all nodes has been used as starting nodes, and the y-axis represents newly visited and overhead nodes also as percentage of n. Figure 4 presents the integral of the functions from Figure 3. We clearly see that after a certain threshold, only few nodes per path are added, while the overhead increases very much. Hence, our improved algorithm should only be applied up to this threshold. As the functions from Figure 4 can be computed while the algorithm is executed, the algorithm can stop automatically when a certain threshold is reached.

Figure 5 presents a detailed view of Figure 4 for $x \leq 0.05\%$. One sees that in this region, which is a more realistic application scenario than to use 1% of all nodes as starting nodes, the algorithm performs much more favorably.

To find out how the algorithm scales with increasing n, we plot the newly visited nodes and overhead for $n = 10^7$ in Figure 6. We see that this function looks as before, and conclude that the algorithm scales well. The same holds for the sum of visited nodes and summed overhead, which is omitted due to space restriction.

4.2 Generator Properties

For reference, we first investigated two functions that are known for a long time, one unbroken and one broken. We started with the Data Encryption Standard (DES) (see e.g. [7]), which has been a standard blockcipher from the seventies till today, although it has been replaced officially by the AES (Advanced encryption standard). The DES is a Feistel cipher with 16 rounds. In each round, one half of the 64-bit codeword is combined with a 48-bit round key by a round function. The round function contains a non-linear part. First, by doubling some of the bits of the code word (so-called expansion permutation), it is increased from 32

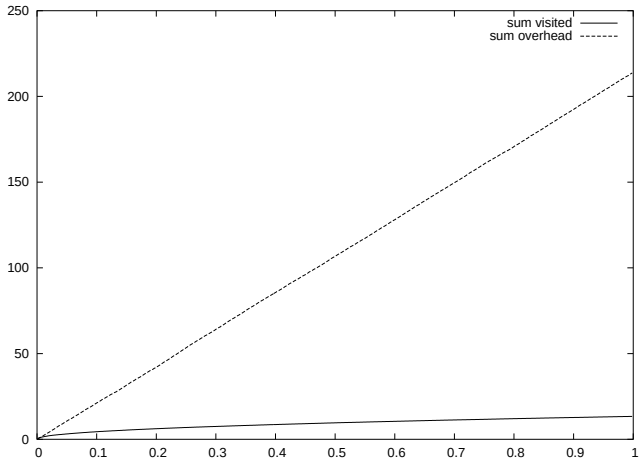

Fig. 4. Sum of visited nodes so far and summed overhead for $n = 10^6$, in percentages of n.

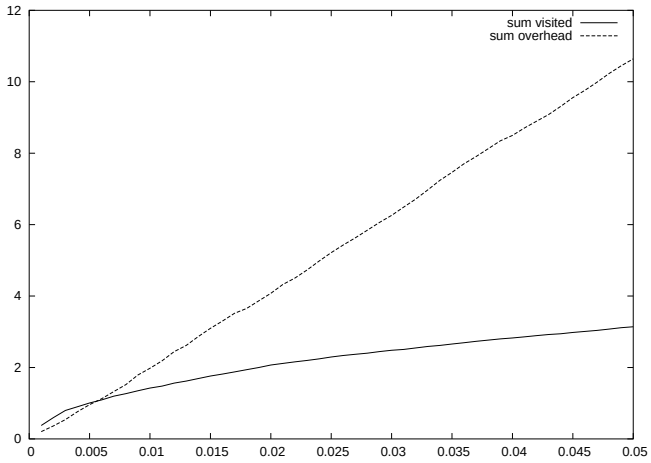

Fig. 5. Detailed view of Fig. 4 for $x \leq 0.05\%$.

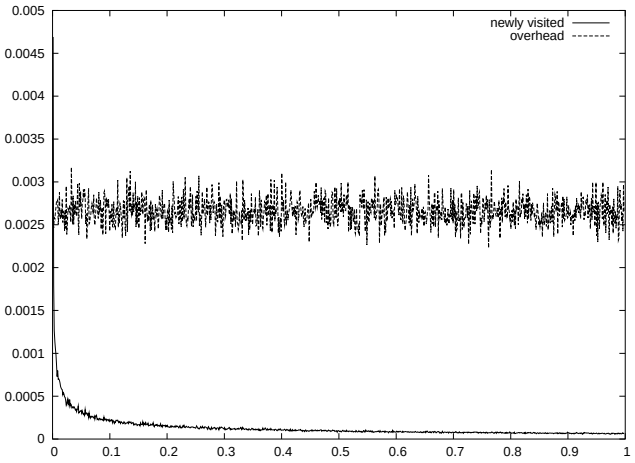

Fig. 6. Newly visited nodes and overhead per starting node for $n = 10^7$, in percentages of n.

to 48 bits. This is followed by the S-box transformation back to 32 bits. There are eight different non-linear S-boxes, each with a 6-bit input and a 4-bit output. As the expansion permutation followed by the S-box transformation is the only non-linear part of DES, and the one protecting DES against differential crypt analysis, we chose this part as a kind of generator transition function. Because of the small size $n = 2^{32}$, this graph could be explored completely. It revealed 11 WCCs (16 would be expected), the largest WCC had a size of $0.8n$ ($0.76n$ would be expected), and an average cycle length of $0.73\sqrt{n}$ ($0.63\sqrt{n}$ would be expected). Hence, this graph looks quite as expected.

Second, we took a pseudo random number generator based on a cellular automaton by Stephen Wolfram [8], which is already known to be predictable [9]. The cellular automaton consists of k linearly connected cells, each being either in state 0 or 1. Each cell's next state is dependent on its own state and the state of its neighbours. Thus, the automaton can assume $n = 2^k$ states. For an automaton with $k = 24$ cells, i.e. $n = 2^{24}$, we revealed 49 WCCs, the largest WCC having a size of $0.94n$, with an average cycle length of $42.5\sqrt{n}$. Also this graph could be explored completely. Compared to the expected values, there are too many WCCs, the largest WCC is much too large, and the longest cycle is much longer than expected.

Finally, we explored the A5/1 generator. It consists of three coupled linear feedback shift registers (LFSR) of lengths 23,22, and 19. Each LFSR has a clock bit. In each cycle, there is a majority vote over the three clock bits, and the registers with clock bits corresponding to the majority are clocked, i.e. each register is clocked in 3 out of 4 cases. The history of A5/1 is quite strange. It has been designed by ETSI (European Telecommunication Standards Institute), but

not been laid open for public scrutiny. We follow the presentation in [1] which refer to other sources that re-engineered the algorithm in a GSM mobile phone and finally got confirmation from GSM about the algorithm. Wagner et. al. also present an attack on this stream cipher, hence the security is not clear, and we felt it to be a good test case.

The state of the generator consists of the concatenated contents of the three LFSRs, thus $n = 2^{64}$. With 326,131 starting nodes we detected 59,661 WCCs. At most 103 starting nodes belonged to one WCC. The largest cycle found had a length of $0.13\sqrt{n}$. A 32-CPU cluster needed one week to compute this result. Hence, the graph looks definitely non-random. Most cycles have length $(4/3) \cdot (2^{23} - 1)$, i.e. they are defined by the period of the longest LFSR, and its frequency of clocking! Similar observations, albeit not with respect to the number of WCCs are made on slide 8 of [10].

5 Conclusions and Future Work

We have presented an algorithm that allows to explore large random graphs better than previous methods. We applied this algorithm to reveal the graph structure of several generators in the cryptographic field. Our next aim is to refine and tune our algorithm, and to explore a larger part of the state graph of A5/1, because the preliminary results indicate a quite unusual structure. Our feeling is that the surprising structure of the A5/1 state graph may also give rise to a further distinguishing attack on the A5/1 output stream.

References

1. Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of A5/1 on a PC. In: Fast Software Encryption Workshop 2000, Springer LNCS (2001) 1–18
2. Eberspächer, J., Vögel, H.J., Bettstetter, C.: GSM — Global System for Mobile Communication. 3rd edn. Teubner-Verlag (2001)
3. Keller, J.: Parallel exploration of the structure of random functions. In: 6th Workshop on Parallel Systems and Algorithms (PASA), VDE (2002) 233–236
4. Heichler, J., Keller, J., Sibeyn, J.F.: Parallel storage allocation for intermediate results during exploration of random mappings. In: 20th Workshop Parallel Algorithms, Strctures and System Software (PARS). GI (2005) 126–134
5. Flajolet, P., Odlyzko, A.M.: Random mapping statistics. In: EUROCRYPT '89, Springer LNCS (1990) 329–354
6. Heichler, J., Keller, J.: A distributed query structure to explore random mappings in parallel. In: 14th Euromicro Conference on Parallel, Distributed and Network-based Processing. IEEE CS (2006) 173–177
7. Schneier, B.: Applied Cryptography. Wiley (1995)
8. Wolfram, S.: Cryptography with cellular automata. In: Proc. Crypto '85, Springer LNCS (1985) 429–432
9. Meier, W., Staffelbach, O.: Analysis of pseudo random number sequences generated by cellular automata. In: Proc. Eurocrypt '91, Springer LNCS (1991) 186–189
10. Gong, G.: ECE 710 Sequence design and cryptography (Fall 2005) lecture slides. http://calliope.uwaterloo.ca/∼ggong/ECE710T4/lec8-ch6b.pdf