FSMlock: Defending against Oracle-based Sequential Logic-Locking Attacks under Output-Corruption Requirements

Ioannis Stavrinos
Information & Comm. Systems Eng. Dept., Univ. of the Aegean, Greece
icsd18207@icsd.aegean.gr

Christos Chalagiannis
Information & Comm. Systems Eng. Dept., Univ. of the Aegean, Greece
icsd15213@icsd.aegean.gr

Emmanouil Kalligeros
Information & Comm. Systems Eng. Dept., Univ. of the Aegean, Greece
kalliger@aegean.gr

***Abstract*—In this paper FSMlock is proposed, which, contrary to the vast majority of the already existing sequential logic-locking techniques that augment or modify a circuit's Finite State Machine (FSM) leaving untouched various normal-operation parts of it, locks the entire FSM using two keys to erase structural traces, and a few simple modifications to the Register-Transfer Level (RTL) description of the circuit. Additionally, FSMlock incorporates a new, efficient, RTL-applicable output corruption (OC) scheme. FSMlock is evaluated against all different kinds of sequential oracle-based attacks, while even after the attacks' application, high OC is obtained. Finally, Verilog RTL descriptions of the ITC'99 benchmark circuits, converted from VHDL to be used in our experiments, are made publicly available.**

***Index Terms*—logic locking, FSM locking, output corruption**

I. Introduction

Logic locking is a well-established hardware security approach that protects a design by inserting into it key-driven hardware structures, which alter its operation in the presence of wrong keys. Depending on whether the locking hardware is placed into the combinational part of a circuit or if its Finite State Machine (FSM) is redesigned for security, logic-locking techniques are divided into combinational and sequential.

A large class of attacks on logic locking are the oracle-based ones. They assume that the attacker is in possession of an activated, functional copy of the circuit, the *oracle*, which can be used to obtain the correct output for any circuit input. The gate-level netlist of the locked circuit is also available to the attacker. Although attacks on combinational locking have been tackled [1], attacks that make use of circuits' sequential operation (sequential ones) also exist. They can be roughly classified into three categories: i) combined structural+functional [2], [3], which try to reveal an obfuscated FSM through structural analysis of the corresponding circuit and subsequent functional analysis of it. ii) Attacks that use a model checker to unroll the circuit and then apply the SAT attack, like KC2 [4] and Fun-SAT [5]. They are very effective, while their computational cost constantly improves [4]–[6]. iii) Approximate attacks, like the Particle Swarm Optimization (PSO)-guided one [7], that try to find approximate keys using low-complexity computations.

Concerning sequential locking techniques, most of them, including HARPOON [8], interlocking [9], state deflection [10], ReTrustFSM [11], Dual Key (DK) Lock [12], augment a circuit's FSM to protect its normal functionality. However, the original FSM can either be separated from its added parts, or access to normal-operation parts of the modified FSM is allowed. Also, the correct reset state is reachable, which, even if it does not belong to the original FSM, serves as an entry point to attacks. The weaknesses of [8]–[10] have been discussed in [11]. In ReTrustFSM, the necessary "lockedFSM" state must meet very strict requirements, and should be located at the deepest FSM stage [11], thus allowing plenty of normal functionality before reaching the FSM's obfuscated part. DK Lock [12] is a two-phase approach that uses one key for circuit activation via up-counting for a specific number of cycles, while the second key is driven to key gates after activation. Nevertheless, in the netlist, the activation counter can be bypassed by forcing the activation signal to its enable value, transforming DK Lock into a simple key-gate insertion scheme.

A few sequential locking schemes that do not rely on FSM augmentation also exist. JANUS [13] controls with keys the D or T state-flip-flop functionality, after splitting the FSM's State Transition Graph (STG) into two parts. A SAT-based attack on JANUS has been described in [11]. Entire FSM hiding has been exploited in [14], but the characteristics of the cellular automata employed for state encoding, were utilized to break the method [15]. Transition obfuscation through state permutation for full FSM hiding was proposed in [16]. However, the fact that states are just a permutation of the original ones, allows the attacker to start from the reset state of the oracle and, by forcing different states in the netlist's state register, to find which one matches the output behavior of that of the oracle (using various inputs). The same process can then be repeated for the next oracle state.

Another important locking aspect is Output Corruption (OC), i.e., the percentage of *bits* per output pattern that differ between correct and faulty operation. In sequential locking, OC is either not discussed, or, usually, relies on some combinational locking scheme (typically key-gate insertion), which should be applied at a different design level and does not guarantee adequate OC.

To address all the above issues, FSMlock is proposed in this paper. Our contributions are: a) entire FSM hiding, including the reset state, b) a new, efficient OC scheme that is applied at RTL, like the locking process. c) The PSO-guided attack has been considered and implemented for the evaluation of FSMlock. d) Verilog RTL descriptions of the ITC'99 benchmarks have been generated, verified and made available [17].

II. The FSMlock Defense

A. FSM Protection

First of all, we assume that the attacker has access to the locked gate-level netlist of the circuit under attack, and knowledge of the locking scheme and the locations of the key inputs. Also, an activated integrated circuit, i.e., oracle, is available, but without scan access. Hence, the attacker can use only the primary inputs and outputs of the activated circuit.

The main idea of FSMlock is to completely hide from the attacker the states of a circuit's FSM and the transitions between them, so that they are not recognizable neither by structural analysis, nor by netlist simulation. To do so, a single key is not

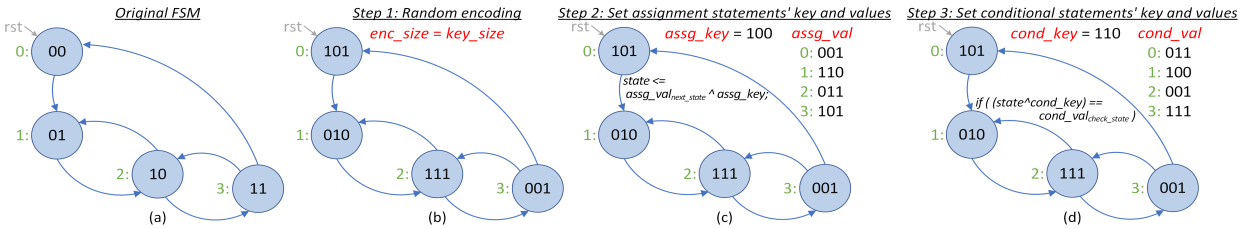

Fig. 1. FSM locking example. (a) Original FSM. (b) Random state encoding after increasing state register's length. (c) Select key to be used in assignment statements and calculate the corresponding value for every state of the FSM. (d) Select key for conditional statements and calculate the respective value for each FSM state.

sufficient as will be explained in a while; instead, two different keys are used. Please note that the utilized keys are constant, and fed to special key inputs of the circuit from a tamper-proof memory, as in the case of combinational locking schemes. An example demonstrating the steps to modify an FSM according to the proposed FSMlock approach, is shown in Fig. 1. Fig. 1(a) depicts the original FSM that consists of four states, the encoding of which is binary (from 00 to 11).

The first step, before locking, is to create a very big state space and, from it, to assign a new, random state encoding to the FSM. This is shown in Fig. 1(b), where the 2-bit encoding of the original FSM has been replaced with a random 3-bit one. Of course, in real implementations, the states' size will increase more so that it becomes equal to the key size. The reason for this re-encoding step is simple: if the locked states' length were small, then a brute-force simulation attack would easily break the locking, as key size would also be small.

Next, to totally hide all state values of an FSM, we should at first consider where they appear in its RTL description. State values can be found in state variables' assignment statements (on their right-hand expressions), and in conditional statements where state variables are checked. We handle the former first, by defining a key specifically for assignment statements ($assg\_key$). Then, for every state-encoding value, we calculate the corresponding assignment value ($assg\_val$), i.e., the value that should be XORed with $assg\_key$ to get the actual encoding value (i.e., for state i, $state\_val_i = assg\_val_i \oplus assg\_key$). For example, in Fig. 1(c), assume that $assg\_key = 100$. The $assg\_val$ of state 3, with actual value 001, will be $assg\_key \oplus 001 = 101$. After $assg\_vals$' calculation, in all right-hand expressions of state-variable assignments, the actual state values are replaced by the corresponding bitwise XOR operation. As an example, the very common: $state <= next\_state;$ assignment in FSM descriptions in Verilog, is replaced by:

$$state <= assg\_val_{next\_state} \hat{} assg\_key;$$

where ^ is Verilog's bitwise XOR operator. This way, the actual FSM's state-encoding values can be neither structurally identified, as they are not directly used in assignments, nor retrieved by simulation, unless the correct key is known. The FSM's RTL description contains and, hence, the implemented circuit hardcodes, the states' $assg\_vals$, which are useless as they cannot be associated with actual states without the $assg\_key$.

At this point, there are two issues that need to be discussed. The first one is that after random state encoding (Step 1), no state value should be equal to all-zeros. In such a case, the corresponding $assg\_val$ would be equal to the $assg\_key$ compromising security. The second is that the described assignment-statements' modification is applied everywhere in the protected FSM, even in the reset-case assignment. Thus, the reset state is unknown to untrusted parties, i.e., the attacker is unaware of the starting point of circuit's operation. This is important, since most attacks consider the circuit's reset state known. We note that in the KC2 attack's publication [4], it is mentioned that "an unknown initial state can also be modeled using additional key-variables". FSMlock protects all states though, not just the initial one, and all the transitions between them; they are all unknown. Moreover, since they depend on the key, they are already associated with it, so the key variables employed by the attack, already include the unknown state information.

The final step to complete state locking is to protect state values in conditional statements (i.e., if, $case$, conditional operator). To do so, the same process as for the assignment statements is followed but with a different key ($cond\_key$). This is essential security-wise, since if we use the same key, then the same state-value set (i.e., $assg\_vals$) as for the assignment statements should be used for the conditional ones. But in this way, state transitions (described in HDL assignments) would be associated with state decoding (HDL conditionals) and, through structural analysis, the STG of a circuit's FSM could be revealed. Mind that if such a thing happens, the attacker could replace the unknown state encoding with their own and get correct FSM operation.

Therefore, we randomly select a new $cond\_key$ and calculate the states' $cond\_val$ set (Fig. 1(d)), exactly as we did with $assg\_key$ and $assg\_vals$. Then, in all conditional statements of an FSM's RTL description, we replace all the occurrences of the state variable with its XOR operation with $cond\_key$, and we check against the necessary value from the $cond\_val$ set. For example, the if statements are modified from:

$$if(state == check\_state)...$$

to: $$if((state \hat{} cond\_key) == cond\_val_{check\_state})...$$

The $case$ statements and the conditional-operator expressions are modified in a similar manner. In the example of Fig. 1, observe that for none of the FSM states, the corresponding $assg\_val$ and $cond\_val$ are equal. As explained, this decouples state transitions from state decoding, effectively removing node labels and edges from an FSM's STG. Although transitions to the same state can be identified, *the attacker cannot associate the origin with the destination states*, due to the different $assg\_val$ and $cond\_val$ sets. This is necessary for revealing the FSM though, as, otherwise, edges cannot be placed in its STG. Thus, the only things that the attacker can learn from the protected FSM, by structural analysis, are the number of states and the number of transitions to every state. If for two values of $assg\_key$ and $cond\_key$, it turns out that, for state i, $assg\_val_i = cond\_val_i$, this can be easily solved by regenerating one of the keys. We also note that for realistic key and state sizes, sets $assg\_val$ and $cond\_val$ will be disjoint.

In Fig. 2, the functional concept of the proposed FSM protection is shown. When $assg\_key$ and $cond\_key$ are correct, the FSM operates as designed. Otherwise, the state register gets unexpected values and none of the actual FSM states is reached.

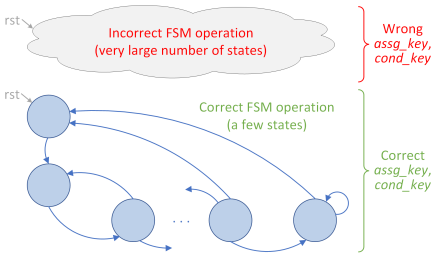

Fig. 2. Functional concept of the FSM protection scheme.

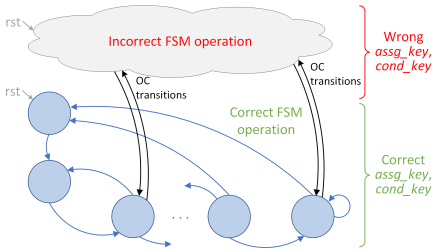

Fig. 3. The complete FSMlock scheme with output corruption (OC) capability.

B. Output Corruption

Output Corruption (OC) is what distinguishes circuit operation with a correct key from that with a wrong key. As has been shown in [7], low OC can allow the approximate utilization of a circuit, even without full-key knowledge. Most sequential logic-locking techniques rely on combinational locking for OC, i.e., on the insertion of key-gates or other combinational locking components. However, this has two disadvantages: a) apart from the RTL code, it requires intervention in the circuit's netlist, which makes locking more complex. b) Since random component insertion is necessary to reduce locking traces that can be used by attacks [1], the resulting OC may not be sufficiently high; random placement provides no guarantee as to how much a component's insertion location affects a circuit's outputs.

To address these issues, we propose an RTL-applicable OC approach, that leverages the functionality of one or very few states of the actual FSM. The idea is, in the existence of a wrong key, to "send" the FSM to one or very few correct-operation states that significantly affect the circuit's outputs, to cause OC. Mind that, in this scenario, transitions between correct-operation states of the FSM are not desirable. What is needed is to "jump" to one of them, exploit its output activity to cause OC, and then "jump" back to the incorrect-operation area. This is shown in Fig. 3, which, additionally to Fig. 2, includes the described OC transitions to two of the FSM's correct-operation states and back.

To implement OC transitions, we make use of the $default$ clause of $case$ statements. As this clause is used to cover all state-register values that do not match any of the $case$ items listed above it, it is suitable for this purpose. The most important issue that has to be addressed in this approach is that the actual encoding value(s) of the selected correct-operation state(s) will be exposed. Although they can be protected against structural analysis, inevitably, in simulation, they will become visible. To prevent revealing of the keys, which would be trivial if the actual encoding of a state, along with its $assg\_val$ and $cond\_val$ were known, we remove the state(s) used for OC from the locking process and we use directly their actual encoding value. In this way, we protect all the rest states and we just reveal a small part of the FSM for OC purposes.

In order not to compromise security, the following rules should be followed when selecting states for OC: a) the fewest possible states that yield a sufficient level of OC should be used; b) the initial state of the FSM should not be selected, so that the starting point of circuit's (correct) operation remains hidden; c) if more than one states are selected, they should be reached in random order; d) edge-connected correct-operation states should be avoided, since they will reveal larger chunks of the FSM's functionality. It is worth noting that if the last rule is followed, the transitions from the OC states back to the incorrect-operation area, are a direct result of the fact that the states following the OC one are locked. Therefore, a wrong key will generate wrong state values for them.

Concerning the common use of the $default$ clause for protection against the occurrence of unspecified states during correct operation (i.e., with correct keys), it can still be realized. Just an additional flag-bit indicating origin from the $default$ clause is needed. In the OC state, this bit will be checked and if it is true, a new encrypted transition (using $assg\_key$) to the appropriate state (usually the reset one) will be performed. As required, with the correct key, the circuit will end up to this state; otherwise, it will return to the incorrect-operation area.

As a final remark, we note that if more than one states are selected for OC purposes, they are alternated in a specific, non-functional order inside the $default$ clause, using a small counter to keep this order.

III. Experimental Results

FSMlock has been implemented in Python. Yosys was first used on the benchmark circuits' RTL descriptions to extract FSM information. For each circuit, this information, the corresponding RTL description, and some extra input data, including the correct-operation states to be used for OC, were fed to the implemented Python utility for applying FSMlock. In circuits that consist of multiple different modules with a separate FSM each, we lock only one of the modules (but all of its instances, with the same two keys), as this was the simplest approach. Regarding OC states, a number of 2 or 4 were utilized. Their selection was made experimentally, by performing a few simulations and observing output activity. In our experiments, the largest ITC'99 benchmarks were used, except for b19 that exhibited constant-zero outputs under random inputs.

Concerning oracle-based sequential attacks, all three categories were considered. The structural+functional one [2], [3] is not applicable to the proposed defense though. This is because, to perform functional analysis, knowledge of the locked FSM's reset state and enumeration of all FSM input values [3] is required. However, in FSMlock, the reset state is unknown, while the circuit's key inputs are driven to the FSM and their large number makes enumeration infeasible.

Against FSMlock, we deployed the KC2 [4] and PSO-guided [7] attacks. To apply KC2, the NEOS open-source tool was utilized. All experiments were executed on a 6-core/12-thread AMD Ryzen5 processor running at 3.9 GHz, and a timeout period of 48 h was set for the attack. *After the attack's termination*, we calculated OC percentages, by simulating every circuit using the returned key, with $5 \cdot 10^5$ random input vectors. Some reset activations were also performed throughout the input-vector sequence. We should mention that the OC calculations are bit- and not pattern-oriented, i.e., the reported percentages represent the average number of bit inversions per output pattern, relative to the corresponding correct one (i.e., mean Hamming distance over the total output-pattern count). The results

TABLE I
KC2 AND PSO-GUIDED ATTACKS RESULTS

Circuit	Key size = 64				Key size = 128			
	Correct key bits (%)		OC (%) after attack		Correct key bits (%)		OC (%) after attack	
	KC2	PSO	KC2	PSO	KC2	PSO	KC2	PSO
b14	47.66	45.16	47.6	47.3	48.05	52.19	47.4	47.2
b15	48.44	45.94	21.9	21.6	48.83	48.44	21.6	21.8
b17	43.75	44.53	10.1	10.0	53.13	49.14	9.8	10.0
b18	55.47	52.03	8.7	9.7	49.22	47.11	8.8	9.7
b20	46.88	46.69	35.6	36.0	53.52	50.63	36.9	35.8
b21	60.16	47.50	48.9	49.0	46.48	52.89	49.1	48.8
b22	53.13	49.84	46.2	46.2	46.88	50.39	46.2	46.2

for the KC2 attack are shown in the "KC2"-labeled columns of Table I. Note that a reported key size of a certain length, means that both $assg\_key$ and $cond\_key$ are of that length.

We mention that for b14 and the circuits that locking is applied to a b14 module (b20, b21 and b22), KC2 terminates quickly (within 1.5 h) but with a wrong key. This can be attributed to the fact that the attack finds a key that leads to oracle-response matching over a small number of unrolling steps, but not for longer time periods [5]. The subsequent equivalence check, using `dsec` ABC command, ends successfully, assuming though the all-zero reset state for the circuits' registers, which is not the case for FSMlock. As for the remaining benchmarks (b15, b17 and b18), the attack timeouts with a wrong key.

As can be seen from Table I, FSMlock practically manages to nullify KC2. Nearly all correct key-bit percentages are around 50% (apart from the 64-bit-key case of b21 that gets to 60.16%), which is equivalent to random-value assignment. As far as OC is concerned, we observe that it is very close to the optimal 50% for b14, b21 and b22, it is very high for b20, acceptable for b15, and somewhat low for b17 and b18. The reason for the lower OC percentages is that many of the corresponding circuits' outputs, maintain a constant value throughout the entire OC simulation.

Concerning the PSO-guided attack, we have implemented it in Python, using the very fast Verilator Verilog simulator. As termination condition, in case of a non-perfect simulation match between the output for a particle (candidate key-pair) and that of the oracle, we set an upper limit of 400 particle generations to evolve. This was based on the observation that the quality of the attack's results, i.e., the attained OC level, was usually saturating well before 400 generations. For the calculation of the correct key-bit percentages and OC, we followed the same process as for KC2, with the difference that we repeated each experiment (attack + OC calculation) five times, as the attack includes random choices. The percentages reported in Table I are the average values of the results of these five experiments.

We can see that the PSO-guided attack has exactly the same, random-like behavior as KC2. This can be attributed to the main features of FSMlock: with a very large state space and without knowledge of the FSM's reset state (i.e., a known correct state to begin with), a random initial key-value group (particle generation) is very difficult to evolve to a high-quality particle generation. Consequently, the OC percentages after the PSO-guided attack, are similarly high as those obtained after KC2.

Finally, we measured the area and delay overheads that FSMlock imposes to the protected circuits. To do so, inside Yosys, we invoked ABC for performing synthesis, utilizing the public FreePDK 45 nm library. For delay measurements we set an upper delay limit of 10000 ps (100 MHz min clock frequency). The derived results are shown in Table II. In terms of area overhead, there is a clear trend of decreasing overhead as circuit size increases. Any discrepancies on this, are due to the existence of multiple locked instances of the same module in circuits b17-b22. However, by taking into account that these benchmarks are of small / medium size, we can deduce that for larger, real-life circuits, area overhead will be very small. Concerning delay, as expected, there are negligible differences between the original and the locked circuits, as the critical path of a circuit does not lie in the control logic (FSM).

TABLE II
AREA AND DELAY OVERHEAD RESULTS

Circuit	Key size = 64		Key size = 128	
	Area ovhd (%)	Delay ovhd (%)	Area ovhd (%)	Delay ovhd (%)
b14	24.9	-3.94	38.82	-7.72
b15	9.47	0.95	21.21	-0.09
b17	9.51	-0.02	22.79	-0.02
b18	8.09	0	16.35	0
b20	10.58	0	14.8	0
b21	10.73	0	14.83	0
b22	3.79	0	6.18	0

IV. CONCLUSIONS

We presented FSMlock, a sequential logic-locking approach against oracle-based attacks that hides the entire FSM of a circuit, by locking its states and transitions using two constant keys. Therefore, neither the reset state of the circuit is available for the attacker to start from, nor parts of the correct FSM operation remain open for them to exploit. FSMlock employs a new RTL-applicable OC approach, which makes use of one or very few correct-operation states that significantly affect the circuit's outputs. By moving to these states from the incorrect-operation FSM area and then back, we get high OC results.

REFERENCES

[1] N. Limaye, E. Kalligeros, N. Karousos, I. G. Karybali, and O. Sinanoglu, "Thwarting all logic locking attacks: Dishonest oracle with truly random logic locking," *IEEE TCAD*, vol. 40, no. 9, pp. 1740–1753, 2021.

[2] T. Meade *et al.*, "Revisit sequential logic obfuscation: Attacks and defenses," in *Proc. of IEEE ISCAS*, 2017.

[3] M. Fyrbiak *et al.*, "On the difficulty of FSM-based hardware obfuscation," *IACR TCHES*, vol. 2018, no. 3, pp. 293–330, Aug. 2018.

[4] K. Shamsi, M. Li, D. Z. Pan, and Y. Jin, "KC2: Key-condition crunching for fast sequential circuit deobfuscation," in *DATE*, 2019, pp. 534–539.

[5] Y. Hu *et al.*, "On the security of sequential logic locking against oracle-guided attacks," *IEEE TCAD*, vol. 42, no. 11, pp. 3628–3641, 2023.

[6] K. Z. Azar *et al.*, "Improving bounded model checkers scalability for circuit de-obfuscation: An exploration," *IEEE TIFS*, vol. 19, 2024.

[7] R. Karmakar and S. Chattopadhyay, "A particle swarm optimization guided approximate key search attack on logic locking in the absence of scan access," in *Proc. of DATE*, 2020, pp. 448–453.

[8] R. S. Chakraborty and S. Bhunia, "HARPOON: An obfuscation-based SoC design methodology for hardware protection," *IEEE TCAD*, vol. 28, no. 10, pp. 1493–1502, 2009.

[9] A. R. Desai *et al.*, "Interlocking obfuscation for anti-tamper hardware," in *Proc. of CSIIRW*, 2013.

[10] J. Dofe and Q. Yu, "Novel dynamic state-deflection method for gate-level design obfuscation," *IEEE TCAD*, vol. 37, no. 2, pp. 273–285, 2018.

[11] M. S. Rahman *et al.*, "ReTrustFSM: Toward RTL hardware obfuscation-A hybrid FSM approach," *IEEE Access*, vol. 11, pp. 19 741–19 761, 2023.

[12] J. Maynard and A. Rezaei, "DK Lock: Dual key logic locking against oracle-guided attacks," in *Proc. of ISQED*, 2023.

[13] L. Li *et al.*, "JANUS: Boosting logic obfuscation scope through reconfigurable FSM synthesis," in *Proc. of IEEE HOST*, 2021, pp. 292–303.

[14] R. Karmakar *et al.*, "A cellular automata guided obfuscation strategy for finite-state-machine synthesis," in *Proc. of DAC*, 2019.

[15] A. Saha *et al.*, "ORACALL: An oracle-based attack on cellular automata guided logic locking," *IEEE TCAD*, vol. 40, no. 12, pp. 2445–2454, 2021.

[16] S. Muzaffar and I. A. M. Elfadel, "Logic locking of finite-state machines using transition obfuscation," in *Proc. of IFIP/IEEE VLSI-SoC*, 2022.

[17] I. Stavrinos and E. Kalligeros, "Verilog RTL descriptions of ITC99 benchmark circuits," https://github.com/ccsl-uaegean/ITC99-RTL-Verilog.