A Novel LBIST Signature Computation Method for Automotive Microcontrollers using a Digital Twin

Daniel Tille[*] Leon Klimasch[*] Sebastian Huhn[†§]

[*]Infineon Technologies AG
85579 Neubiberg, Germany
{Daniel.Tille,Leon.Klimasch}@infineon.com

[†]University of Bremen, Germany
huhn@uni-bremen.de

[§]Cyber-Physical Systems
DFKI GmbH
28353 Bremen, Germany

***Abstract*—LBIST has been proven to be an effective measure for reaching functional safety goals for automotive microcontrollers. Due to a large variety of recent innovative features, every customer can adjust LBIST settings in a way that fits their use case. The downside of these user-defined configurations is the handling of their golden signatures: Traditionally, they can be computed only with access to the gate-level netlist. This is typically not possible for MCU customers because a netlist contains protected IP, which cannot be disclosed to third parties.**

This paper proposes a digital twin of the LBIST functionality that can overcome this drawback. It is an executable model that can be delivered together with the product. As a result, for the first time, a customer can compute a golden signature without knowledge of the netlist or other support of the supplier. We prove the efficacy of the digital twin in an industrial environment on an automotive microcontroller.

I. Introduction

In the last two decades, the digitization has heavily impacted the automotive industry. Nowadays, the majority of the functionality is implemented using *Microcontrollers* (MCUs) – even safety-critical functions such as airbag control, power steering, and braking systems. As a consequence, the correct functionality of these chips is more important than ever since malfunctions could lead to large collateral damage. It is not sufficient anymore to just test circuits for defects after manufacturing. Instead, functional safety standards, such as the ISO26262 [1], demand so-called in-field tests that regularly check an MCU for defects (e.g. during each power-up).

Logic Built-In Self-Test (LBIST) is a state-of-the-art structural in-field test method, which is widely used for automotive MCUs [2], [3]. This approach applies a deterministic sequence of pseudo-random test patterns to the circuit logic. The test responses are continuously compacted into a *signature*, which can be evaluated after the end of the test. Since the circuit behavior is deterministic during LBIST, there is an unambiguous *golden* signature for a correct chip for each specific test sequence. This golden signature can be computed by *Gate-Level Simulation* (GLS). Usually, there is a default LBIST configuration whose golden signature is determined and provided in the MCU's datasheet.

At the same time, state-of-the-art automotive MCUs employ LBIST controllers which are highly configurable. The user can program various parameters such as the seed of the pseudo-random sequence, the number of test cycles, or advanced power reduction measures [4]. Each *Electronic Control Unit* (ECU) family, i.e., the system that integrates MCUs, can use a distinct LBIST configuration in order to satisfy its specific system requirements (w.r.t. runtime, power, ...). Since each configuration yields a different golden signature, this leads to some practical challenges.

The golden signature for a specific LBIST configuration is required in order to evaluate the correctness of the test result. However, MCU customers that program LBIST applications do not have the capabilities to compute the golden signature of a user-defined configuration. They do not have access to the MCU's gate-level netlist, which is required for executing GLS, because MCU suppliers cannot disclose it. The protection of (own and third-party) *Intellectual Property* (IP) as well as the need for confidentiality of hardware security implementations prevent a disclosure.

There are currently three practical possibilities to solve this dilemma. Firstly, the LBIST execution can be limited to a small set of default configurations for which the MCU designer provides the respective golden signatures in the datasheet. This option was common practice in the past when the LBIST application was straightforward and affected only a few use cases. Nowadays, however, there are too many advanced LBIST functions and too many different application scenarios. Therefore, MCU customers no longer accept such a limitation.

Secondly, the MCU designer computes and provides golden signatures for user-defined configurations on demand. While this was common practice in the past, the growth of the automotive market with many new customers and the significant expansion of LBIST applications render this option impractical.

Thirdly, the golden signature can be experimentally determined on a *known good die*. This is a practical solution for the prototyping phase of a product. However, in an environment that demands high quality, some MCU customers are not satisfied with this experimental approach. Also, if the product needs to be certified according to functional safety standards, some auditors might require the golden signature to be computed independently.

In this paper, we overcome this challenge by introducing a **digital twin** of the LBIST functionality. It is an executable model that already integrates all relevant circuit information. That means, the novelty over existing simulation approaches is that no separate netlist is required because it is already an integral part of the program itself. The digital twin can be provided together with the product. Consequently, for the first time, MCU customers can compute golden signatures of their user-defined LBIST configurations without further information or support from the MCU supplier. This significantly improves turn-around time of ECU development and gives the customer a higher degree of freedom when choosing an LBIST configuration.

The contribution of this paper includes

- a novel concept of a digital twin that emulates the LBIST functionality without possession of the gate-level netlist,
- a fully-automated framework for extracting and compiling a digital twin from an arbitrary circuit,
- a prototype implementation of the extraction method, and
- an experimental evaluation on an automotive MCU benchmark.

The remainder of the paper is structured as follows. The next section presents the concept of the digital twin. Section III shows how the digital twin is automatically generated. Afterwards, Section IV gives an overview of our prototype implementation. Experimental results are reported in Section V. Finally, conclusions are drawn in Section VI.

II. Novel Digital Twin

In this section, we propose the new digital twin concept. The main idea of our approach is to provide an executable model that "emulates" the LBIST functionality. (For more details of LBIST, we refer to [5].) In particular, a C++ program computes all relevant LBIST steps like they are executed on a chip. This includes the pseudo-random pattern generation using the *Linear Feedback Shift Register* (LFSR), shift and capture operations, and the determination of the signature in the *Multiple-Input Shift Register* (MISR). The input of the program is an arbitrary LBIST configuration which then controls and influences its concrete execution by providing the content of *Special Function Registers* (SFRs), such as the number of shift cycles. The output is the golden signature for this specific configuration.

The key factor of such a model is the correct program architecture. This is because a C++ program executes all instructions sequentially, whereas in hardware, all operations are performed simultaneously. In the following, we describe how such a model can be obtained.

A. Combinational Logic

Representing the combinational part of a circuit in C++ instructions is quite straightforward. Each combinational sub-circuit with n inputs and m outputs computes a Boolean function[1]

$$f : \mathbb{B}^n \rightarrow \mathbb{B}^m. \qquad (1)$$

[1]For the sake of simplicity, we only consider Boolean logic in the following. An enhancement to a multi-valued logic (e.g. including X and Z values) increases the complexity but is possible through a Boolean encoding [6].

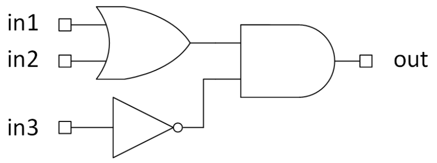

Figure 1: Example circuit

TABLE I: Overview of the variables stored for each FF

Variable	Description
`FF_Q`	current value
`FF_D`	next value from D-input
`FF_TI`	next value from TI-input
`FF_TE`	indicates active scan-enable signal
`FF_CLK`	indicates clock pulse
`FF_R`	indicates reset pulse

Since a combinational circuit has on gate-level a zero-delay timing and does not contain (combinational) feedback loops, it is relatively easy to find a C++ statement that computes its Boolean function. Figure 1 gives an example of a circuit with three inputs and one output. The associated C++ statement is as follows

```
out = (in1 || in2) && !in3;        (2)
```

where `in1`, `in2`, `in3`, and `out` are Boolean variables. (Variables will be explained in the next section). That means the output value is determined by Boolean operations – the same operations as given in the gate-level netlist – applied to the input values. In the case of a combinational sub-circuit with m outputs, there will be m such statements.

B. Sequential Elements

In hardware, *Flip Flops*[2] (FFs) store values from one clock cycle to the next one. We use Boolean variables to implement this behavior in our C++ program. Table I gives an overview of all information that is stored for each FF. The *current value* of a FF is stored in the variable `FF_Q`. The *next value*, which is propagated through the combinational logic to the D-input, is stored in `FF_D`. For *Scan FFs* (SFFs), there are two further inputs, TI and TE, which represent the *scan-in* and *scan-enable* input, respectively. The variables that implement the clocking scheme are explained in the next section.

Figure 2 shows the netlist of the previous example where the inputs and output of the combinational circuit are modeled with FFs. The output of *FF3* is fed back to one of the combinational circuit's inputs. The corresponding C++ statement is as follows:

```
FF3_D = (FF3_Q || FF1_Q) && !FF2_Q;    (3)
```

That means, the value of the output-FF in the next clock cycle, i.e., its D-input-value, is computed through a series of Boolean operations applied to the current values of the input-FFs, i.e., their Q-output-values.

[2]In this work, we consider only flip flops as sequential elements due to the advantages of full-scan designs for LBIST. However, modeling other elements, such as latches and memories, is possible with this approach.

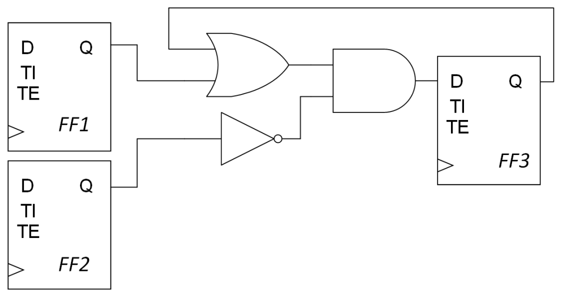

Figure 2: Example circuit with sequential elements

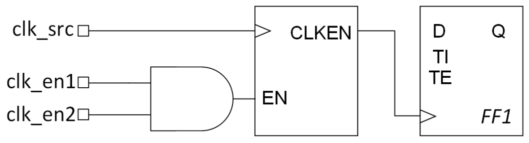

Figure 3: Example for a clock signal controlled by a clock gate

The sequence of the statements for all FFs in a circuit is irrelevant since all values are stable and a D-value depends on the logic in its fan-in cone only, as given within one clock cycles in this abstraction level. Furthermore, the scan path is modeled according to the same principles: The `FF_TI` values are determined based on the `FF_Q` values of the preceding SFFs.

Primary Inputs (PIs) to the chip are also modeled by variables. Since inputs are constrained to a constant value in an LBIST context (otherwise, there would be no stable signature), the corresponding variables are also assigned a constant value.

Finally, also the LFSR and the MISR are modeled using Boolean variables.

C. Clocking Scheme

In hardware, a FF takes over the value at its D-input with a rising clock edge. However, our program does not explicitly model the "continuous" clock signal. Instead, we abstract and just encode the following essential information.

Firstly, within one clock cycle, all values are stable and the actual timing information is irrelevant to the LBIST context. Therefore, we always take into account a complete clock cycle as one discrete entity. In the program context, this is accomplished as one loop iteration where the C++ statements explained above are all executed. Secondly, we model the clock-enabling conditions for each FF using the `FF_CLK` variable. This is done during each clock cycle, i.e., at the beginning of each loop. That means `FF_CLK` is true if, and only if, the FF receives a clock edge in this specific cycle.

Figure 3 shows an example. We see a FF whose clock input is connected to a clock source that provides a scan clock. As it is common practice, the clock input is not directly connected to the source, but there is a *clock gate* that can block the clock signal. The enable pin is connected to some arbitrary enabling logic. The *clk_en1* and *clk_en2* inputs that we see in this example are an abstraction. This enabling logic can usually be complex or even depend on one specific parameter of the user-defined LBIST configuration.

In our approach, we only model the enabling logic when computing the `FF_CLK`, which means for this example:

$$\texttt{FF1\_CLK = clk\_en1 \&\& clk\_en2;} \tag{4}$$

The *clk_src* input is used to identify the clock domain. The signal itself can be disregarded. In the case of a hierarchical instantiation of clock gates, the complete enabling logic cone is taken into account for the computation of the `FF_CLK`.

The scan-reset network is generated similarly. Due to page limitations, we will not discuss details here.

D. Program Architecture

This section gives a brief and high-level overview of the architecture of the C++ program. We abstract from most implementation details in order to maintain readability.

Algorithm 1 presents an outline of the main function of the C++ program. During `init()`, all parameters are read and variables that reflect SFRs (such as the number of LBIST cycles) as well as PIs are set. The outer for-loop (line 3) realizes an LBIST cycle of shift and capture. The number of iterations can be specified through the program parameters. Each iteration of the inner for-loops (lines 4 and 12) corresponds to one clock cycle where one shift or one capture cycle, respectively, is being emulated. At the end of the program, the current state of the MISR contains the golden signature and is printed.

Algorithm 1: Main function of the C++ program

```
1  int main(){
2    init();
3    for (i=0;i<SFR.lbist_cycles;i++){
4      for (j=0;j<chain_length;j++){
5        update_lfsr();
6        shift();
7        update_misr();
8      }
9      if (SFR.scan_reset){
10       reset();
11     } else {
12       for (k=0;k<SFR.capture_cycles;k++){
13         capture();
14       }
15     }
16   }
17   print_misr();
18 }
```

In Algorithm 2, we depict how one capture cycle is modeled in C++. Firstly, the D-values for all FFs are determined (see Eq. 3). Secondly, all CLK-values are determined (see Eq. 4). At the end of the capture cycle, the Q-value is updated for each FF in the circuit, if it receives a clock-edge.

While the skeleton of the program is independent of the actual circuit, the functions which compute the D- and CLK-values (along with the scan-chains in the shift function, which

Algorithm 2: Capture function of C++ program

```
void capture(){
  compute_D_values();
  compute_CLK_values();
  for (i=0;i<FF.size();i++){
    if (FF[i].CLK){
      FF[i].Q = FF[i].D;
    }
  }
}
```

are not reported here) implement actual circuit behavior. This is how the program incorporates netlist knowledge into the digital twin. Therefore, such a program determines the signature for one specific MCU product. For a new design (even if it is just a minor change in the netlist functionality), a new C++ program is required.

E. Advanced LBIST Features

Due to page limitations, we can only briefly mention that our model is compliant with all major advanced LBIST features. Power reduction measures, both for shift and capture, are present in the circuit structure and are therefore taken into account during the generation of the model. The same is true for *Q-gating* [7] and *observe scan technology* [8].

Re-seeding [9] or *bit-flipping* techniques [10] can be implemented by enhancing the `update_lfsr()` function. More sophisticated methods such as *full-scan LBIST* [11], that stores a set of LBIST seeds in an MCU's memory, can be realized as well: an external file containing this set of seeds is processed during `init()` and provides this data for re-seeding.

The application of at-speed LBIST makes the clocking scheme more complex, especially if there are several different clock domains. However, it is generally possible to enhance the model accordingly.

X-tolerant LBIST [12] and X-canceling MISRs [13] are not implemented by the current model. However, with an enhancement to a multi-valued logic containing X-values and the fact that all masking logic is part of the circuit and, hence, part of the program, the integration of such techniques is generally feasible.

III. Automatic Model Extraction

In this section, we show how the C++ model can be automatically generated for an arbitrary netlist. The fundamental idea of our extraction algorithm is, first, to have a library of primitive Boolean functions that represent the functions of all possible gate types in C++. Secondly, a structural circuit traversal method recursively generates C++ statements based on these primitive functions.

A. Gate Library Construction

In practice, a gate-level netlist is synthesized based on the gates contained in a *standard cell library*. The first step of our extraction method is to generate a Boolean function description for each of the individual gate types in such a library.[3]

TABLE II: Examples of gate library functions

Gate	Function
and	`bool AND2(bool a,bool b){return a&&b;}`
or	`bool OR3(bool a,bool b,bool c){return a||b||c;}`
not	`bool NOT(bool a){return !a;}`
mux	`bool MUX(bool a,bool b,bool s){return (s)?b:a;}`

Table II presents a few illustrative examples of functions in the library. The function `AND2` receives the two inputs of an AND gate as a parameter and returns their conjunction. The next two functions show that it is also possible to implement functions with more respectively fewer than two inputs. The final example presents the Boolean function of a multiplexer which returns the value of `b` if the `s` input is true, and `a` otherwise.

Some standard cell libraries contain special gates with more than just one output. Their Boolean function is represented by multiple library functions – one for each output. *Full adder* gates are an example: There are two corresponding library functions, one computing the sum-output and one computing the carry-out-output.

B. Netlist Conversion

The generation of the actual C++ model is explained in the following. The conversion algorithm works on the gate-level netlist and has as output C++ statements. In order to accomplish this, we follow the basic principles explained in Section II-B.

As a first step, all FFs and PIs are assigned Boolean variables (see Table I). In order to generate a C++ statement that computes the `FF_D` value of a FF, a recursive circuit traversal procedure is called for its predecessor gate. An abstract pseudo code[4] is presented in Algorithm 3. When the recursion reaches a terminal gate, i.e., a FF or a PI, it prints the associated variable. Otherwise, it prints the primitive Boolean function introduced above and calls itself recursively for each predecessor gate in order to determine the function's parameters. This way, the C++ statement is created that computes the *next value* of a FF depending on the *current values* of the FFs in its input cone.

Let us revisit Figure 2 for an illustrative example. In order to generate the statement that computes `FF3_D`, the trace procedure is called for the AND gate. This prints the primitive function `AND2` and calls the procedure recursively for the OR gate. Again, the corresponding primitive function is printed and the procedure is recursively called for FF3. This time we meet the termination criterion and the variable `FF3_Q` is printed. The program continues with the next calls until all terminal nodes are finally reached. The final statement looks as follows:

```
FF3_D=AND2(OR2(FF3_Q,FF1_Q),NOT(FF2_Q));
```

[3]There are usually multiple standard cells for each gate type, e.g., representing different driver strengths. We only consider one representative Boolean function for the complete equivalence class of gates of the same type.

[4]We explain here only basic functionality and do not consider all necessary measures to produce syntactically correct code, e.g., correct parentheses.

Algorithm 3: Netlist traversal procedure (Tcl script to generate C++ program)

```
proc trace (gate){
  if (gate.type == TERMINAL){
    print gate.var_Q
  } else {
    print gate.primitive_function
    foreach pred_gate in gate.inputs
      trace (pred_gate)
  }
}
```

Due to its recursive nature, this implementation of the C++ statement looks different than the one presented in Eq. 3. However, when resolving the functions, it is easy to see that they are functionally equivalent.

The tracing algorithm is applied to all FFs in the circuit. This creates the `compute_D_values()` function from Algorithm 2. Similar techniques are applied to the input cones of all other input-pins (see Table I) in order to generate statements that compute the complete circuit functionality, including scan chains, clock control, and the reset network.

C. *Conversion Improvement*

The recursive algorithm explained above scales with the number of paths in the circuit. This can be squared in the number of gates in the worst case. In order to decrease this complexity, we propose an optimization. Each time a *Fanout-Free Region* (FFR) is traversed for the first time, the resulting sub-statement is stored. This is easily possible due to the recursive nature of the traversal algorithm. When the FFR is re-visited, the sub-statement can be returned immediately. This reduces the complexity such that the algorithms scales linearly with the number of gates.

IV. Prototype Implementation

The proposed method has been implemented in Infineon's environment. This work's focus has been on providing a proof-of-concept for a realistic scenario. That means, in the current stage, reaching an optimal runtime was not our priority.

The construction of the gate library (see Section III-A) is straightforward and accomplished using a Python script. This script traverses the relevant standard cell library files and translates the description of the Boolean functions into C++ functions.

The generation of the C++ model (see Section III-B) follows a two-step approach: firstly, since the skeleton of the program is independent of the actual netlist, it can be obtained from a pre-generated template. Secondly, the actual netlist-dependent functions, such as `compute_D_values()`, are generated through a netlist traversal as explained in Algorithm 3. For the sake of simplicity, we decided to implement these algorithms on top of a commercial simulation tool. This is only to prove general feasibility – a final product does not require such a third-party tool.

TABLE III: Benchmarks

Benchmark	gates	chains	max chain length
LEON3	55,416	20	85
TriCore	1,016,308	650	101

TABLE IV: Generation of Digital Twin

Benchmark	runtime [m:s]		file size [kB]	
	generation	compile	source	binary
LEON3	3:48	6:45	6,042	4,429
TriCore	45:00	78:32	26,569	2,482

A. *Proof-of-Concept*

We executed the prototype on two benchmarks: the publicly available LEON3 processor [14] and a TriCore CPU core which is part of Infineon's AURIX automotive MCU [15] (details will be reported in the next section). Both benchmarks were synthesized using state-of-the-art automotive CMOS technology. The LBIST scheme consists of one clock domain with one synchronous scan clock. The designs are designed to be X-free, i.e., they can be modeled using Boolean logic only.

We generated the digital twin for both benchmarks and determined their golden signatures for four different shift cycle configurations. Afterwards, we compared them to the simulation results and validated the signature match.

B. *Safety and Security Discussion*

From a functional safety point of view, there are no relevant safety violations to consider. If the digital twin provides a wrong golden signature, the mismatch will be apparent immediately. The likelihood that an incorrect golden signature will mask a defect in an MCU is negligible: for the implemented 32-bit MISR, it is equal to 2^{-32}. This is the same probability that is usually accepted, e.g., for aliasing [16].

Security considerations and protection of intellectual property are of serious concern – after all, this is the reason why the netlist cannot be provided to the customer in the first place. Our approach, however, does not pose any relevant threat: firstly, the C++ model is a very coarse abstraction of the MCU. It covers only the LBIST functionality. The *mission mode*, especially its timing information, is not extracted. Secondly, for critical circuit parts, such as hardware implementations of encryption logic, there is the possibility of applying code obfuscation [17] during a post-process. This can effectively prevent reverse engineering [18].

V. Experimental Evaluation

Table III reports details of the used benchmarks. We conducted our experiments on a Linux machine with 2.6 GHz and 512GB RAM. We used the gcc compiler in version 4.8.5.

For our experiments, we used a state-of-the-art automotive CMOS standard cell library. It contains 159 different gate types in total; this is also the number of functions in our gate library. The generation of this library requires a runtime of about one second and is a one-time effort.

TABLE V: Execution of Digital Twin

Benchmark	runtime [s]				
	shift cycle	capture cycle	LBIST cycles		
			100	1,000	10,000
LEON3	∅ 6.25e-5	∅ 0.063	6.4	94.0	1,080.0
TriCore	∅ 2.8e-3	∅ 0.025	1.7	21.1	210.3

Table IV presents information about the generation of the C++ program. Columns *generation* and *compile* give the runtimes required for the complete generation of the C++ file and for compiling it, respectively. Columns *source* and *binary* report the file size of the C++ source file and the resulting binary, respectively. We enabled full compiler optimization. If code obfuscation is employed, the compile time and resulting file size will increase.

The fact that we use a commercial tool as the back-end for our generation framework has a significant negative impact on the runtime required for generating the model. A native implementation with data structures that are optimized for this specific use case would yield significantly better results. At the same time, although a low runtime for the C++ program generation is obviously desirable, it is not crucial. Firstly, this generation has to be executed only once for a product. Secondly, this step is not on the critical path of the project: it can be started directly after tape-out, but has to be finished only when customer samples are shipped (which is usually in the order of months after tape-out).

Table V reports the experimental results of executing the digital twin. It gives the average runtimes for one single shift and capture cycle, i.e., the time required for executing the `shift()` and `capture()` functions, respectively. Afterwards, it presents the runtime required for the complete program run for different numbers of LBIST cycles.

Since the program is executed on the customer side, a low runtime matters. This can be achieved through further optimization: The current implementation employs a straightforward translation of the gate-level logic into C++ statements. The number of operations, and with this the required runtime, can be reduced by optimizing the circuit logic. There are multiple optimization approaches available that potentially achieve such an optimization of the circuit logic, e.g., [19], [20].

VI. Conclusions

In this paper, we presented a novel digital twin that allows for emulating an LBIST run. This enables MCU customers, for the first time to independently compute the golden signature for their user-defined LBIST configurations without access to the gate-level netlist or requiring additional support from the supplier. Furthermore, we provided a proof-on-concept of our method demonstrating its feasibility for state-of-the-art automotive MCUs.

Several conceivable directions exist in which the digital twin model could be enhanced. Currently, customers do not have any knowledge about the fault coverage of their user-defined configuration. However, this information is invaluable during the preparation of the *safety case*. A new fault simulation technique using the digital twin could provide an estimation. Another application could be the LBIST diagnosis. Our model can be enhanced with techniques such as [21]–[23]. The resulting approach could significantly speed up the search for a failure in an ECU when an LBIST run fails in a car. MCU customers that employ such a diagnostics method would have a significant advantage over their competition.

References

[1] "ISO26262: Road vehicles functional safety-part 5," 2018.

[2] F. Reimann, M. Glaß, J. Teich, A. Cook, L. Rodríguez Gómez, D. Ull, H.-J. Wunderlich, U. Abelein, and P. Engelke, "Advanced diagnosis: SBST and BIST integration in automotive E/E architectures," in *Design Automation Conference*, 2014.

[3] T. McLaurin, "Periodic online LBIST considerations for a multicore processor," in *IEEE International Test Conference in Asia*, 2018, pp. 37–42.

[4] D. Czysz, M. Kassab, X. Lin, G. Mrugalski, J. Rajski, and J. Tyszer, "Low power scan shift and capture in the EDT environment," in *IEEE International Test Conference*, 2008, pp. 1–10.

[5] V. D. Agrawal, C. R. Kime, and K. K. Saluja, "A tutorial on built-in self-test. I. principles," *IEEE Design & Test of Computers*, vol. 10, no. 1, pp. 73–82, 1993.

[6] R. K. Brayton and S. P. Khatri, "Multi-valued logic synthesis," in *International Conference on VLSI Design*, 1999, pp. 196–205.

[7] X. Lin and J. Rajski, "Test power reduction by blocking scan cell outputs," in *IEEE Asian Test Symp.*, 2008, pp. 329–336.

[8] N. Mukherjee, D. Tille, M. Sapati, Y. Liu, J. Mayer, S. Milewski, E. Moghaddam, J. Rajski, J. Solecki, and J. Tyzer, "Time and area optimized testing of automotive ICs," *IEEE Transaction on VLSI Systems*, vol. 29, no. 1, pp. 76–88, 2021.

[9] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An efficient BIST scheme based on reseeding of multiple polynomial linear feedback shift registers," in *International Conference on CAD*, 1993, pp. 572 – 577.

[10] H.-J. Wunderlich and G. Kiefer, "Bit-flipping BIST," in *International Conference on CAD*, 1996, pp. 337 – 343.

[11] D. Czysz, M. Kassab, X. Lin, G. Mrugalski, J. Rajski, and J. Tyszer, "Full-scan LBIST with capture-per-cycle hybrid test points," in *IEEE International Test Conference*, 2018, pp. 1 – 9.

[12] P. Wohl, J. A. Waicukauski, G. A. Maston, and J. E. Colburn, "XLBIST: X-tolerant logic BIST," in *IEEE International Test Conference*, 2018, pp. 1–9.

[13] J.-S. Yang and N. A. Touba., "X-canceling MISR architectures for output response compaction with unknown values," *IEEE Transaction on CAD of Integrated Circuits and Systems*, vol. 31, no. 9, pp. 1417–1427, 2012.

[14] "LEON3 processor," https://www.gaisler.com/index.php/products/processors/leon3.

[15] "AURIX™ TriCore™," https://www.infineon.com/aurix.

[16] T. W. Williams, W. Daehn, M. Gruetzner, and C. W. Starke, "Aliasing errors in signature analysis registers," *IEEE Design & Test of Computers*, vol. 4, no. 2, pp. 48–57, 1987.

[17] C. S. Collberg, C. D. Thomborson, and D. W. K. Low, "Obfuscation techniques for enhancing software security," US patent no. 6668325B1, 1998.

[18] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Information Hiding*, ser. Lecture Notes in Computer Science, vol. 6958, 2011, pp. 270–284.

[19] N. Eén, A. Mishchenko, and N. Sörensson, "Applying logic synthesis for speeding up SAT," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 272–286.

[20] D. Tille, S. Eggersglüß, R. Krenz-Bååth, J. Schloeffel, and R. Drechsler, "Improving CNF representations in SAT-based ATPG for industrial circuits using BDDs," in *IEEE European Test Symp.*, 2010, pp. 176–181.

[21] I. Pomeranz and S. M. Reddy, "On dictionary-based fault location in digital logic circuits," *IEEE Transaction on Comp.*, vol. 46, no. 1, pp. 48–59, 1997.

[22] A. Cook, M. Elm, H.-J. Wunderlich, and U. Abelein, "Structural in-field diagnosis for random logic circuits," in *IEEE European Test Symp.*, 2011, pp. 111–116.

[23] D. Tille, B. Gottinger, U. Pfannkuchen, H. Graeb, and U. Schlichtmann, "On enabling diagnosis for 1-pin test fails in an industrial flow," in *ASP Design Automation Conference*, 2018, pp. 233–238.