In-Memory SAT Evaluation on RRAM Crossbars: From Exact Verification to Sampling-Based Falsification

Fatemeh Shirinzadeh* fatemeh.shirinzadeh@dfki.de
Abhoy Kole* abhoy.kole@dfki.de
Kamalika Datta*‡ kdatta@uni-bremen.de
Saeideh Shirinzadeh*† saeideh.shirinzadeh@dfki.de
Rolf Drechsler*‡ drechsler@uni-bremen.de

‡*Institute of Computer Science, University of Bremen, Germany*
**German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany*
†*Fraunhofer Institute for Systems and Innovation Research (ISI), Karlsruhe, Germany*

***Abstract*—In-memory computing with RRAM crossbar architectures provides a massively parallel substrate for accelerating Boolean reasoning. In this work, we introduce an in-memory SAT-based verification framework that maps Boolean formulas directly onto crossbar arrays. While deterministic CNF-based evaluation enables exact SAT/UNSAT decisions, exhaustive exploration of all input assignments leads to prohibitive latency, even when executed in parallel in-memory. To address this limitation, we propose a sampling-based in-memory falsification method that explores the input space through randomized evaluation directly on the crossbar. The proposed approach focuses on detecting functional discrepancies between designs by identifying counterexamples rather than proving equivalence exhaustively. As soon as a mismatch is observed, non-equivalence is established, enabling rapid bug detection with significantly reduced computational effort. Experimental results on standard benchmark suites demonstrate that the proposed method achieves substantial reductions in execution latency and crossbar utilization compared to exhaustive evaluation, while effectively detecting non-equivalent designs. These results highlight the potential of sampling-based in-memory falsification as a scalable pre-verification strategy for RRAM-based computing systems.**

***Index Terms*—RRAM crossbar, in-memory computing, MAC operation, equivalence checking**

I. Introduction

In-memory computing *(IMC)* based on resistive random-access memory (RRAM) has emerged as a promising paradigm to mitigate the performance and data-movement bottlenecks caused by data movement in conventional von Neumann architectures. By performing computation directly within memory arrays, IMC eliminates costly data movement and enables massive parallelism, making it particularly attractive for logic-in-memory applications and large-scale Boolean processing [1]–[6].

While substantial progress has been made in mapping logic and arithmetic circuits onto RRAM crossbars, verification of in-memory implementations remains largely dependent on conventional, software-based SAT solvers. These solvers operate off-chip and do not exploit the computational capabilities of IMC architectures. As a result, verification becomes a scalability bottleneck, especially for large designs and equivalence checking tasks, where unsatisfiable instances are prevalent.

In contrast to conventional software or FPGA-based implementations, where logic evaluation proceeds sequentially, the in-memory approach performs computation directly within the memory fabric. This enables constant-time evaluation per logic level and avoids repeated data transfers between memory and processing units, which are major sources of latency and energy consumption.

To address this gap, we first introduce an in-memory SAT evaluation approach that maps Boolean formulas in conjunctive normal form *(CNF)* directly onto RRAM crossbar arrays. In this formulation, the logical structure of the SAT problem is encoded into the crossbar, enabling clause evaluation and aggregation to be performed using multiply-and-accumulate *(MAC)* operations directly within the memory fabric. This establishes a deterministic baseline for in-memory verification, where SAT/UNSAT decisions can be computed without relying on off-chip solvers.

Although this enables efficient in-memory SAT evaluation, a fundamental challenge remains: deterministic evaluation requires exhaustive exploration of the input space, leading to exponential complexity even under crossbar-level parallelism. This fundamentally limits scalability for large designs.

To address this limitation, we propose a sampling-based in-memory falsification method that explores the input space through randomized evaluation directly on the crossbar. Instead of exhaustively proving equivalence, the proposed approach focuses on detecting functional discrepancies between designs by identifying counterexamples. By exploiting the asymmetry of equivalence checking, where a single counterexample is sufficient to prove non-equivalence, the method

enables rapid bug detection with significantly reduced computational effort.

Unlike conventional random simulation, the novelty of this work does not lie in the use of sampling itself, but in the *hardware–algorithm co-design*. Specifically, Boolean verification is reformulated into MAC-compatible operations that can be executed directly on RRAM crossbars, enabling massive parallelism at the hardware level. This tight integration allows randomized evaluation to fully exploit the computational model of in-memory architectures, leveraging massive parallelism and minimal data movement. As a result, verification shifts from an off-chip, sequential process to an in-memory parallel computation paradigm.

The benefits of the proposed approach are particularly pronounced for large CNF instances with high clause-level parallelism, where crossbar columns can be fully utilized.

The proposed framework is evaluated using standard benchmark suites, demonstrating scalability across a wide range of circuit sizes. The results highlight the trade-off between completeness and computational cost, with the CNF-based method providing exact verification and the sampling-based method enabling efficient detection of non-equivalence in RRAM-based IMC architectures.

The main contributions of this work are as follows:

- We present an RRAM-based in-memory verification framework that maps CNF Boolean formulas directly onto crossbar arrays and enables exact verification through MAC-compatible in-memory execution.
- We introduce a sampling-based in-memory falsification approach for scalable equivalence checking, complemented by a deterministic CNF-based verification baseline.
- We provide analytical and experimental evaluations on standard benchmark suites, demonstrating significant reductions in verification cost and crossbar utilization while maintaining effective mismatch detection.

By integrating verification directly into the memory fabric, the proposed framework enables efficient execution of exact and sampling-based verification tasks on RRAM crossbars. The remainder of this paper is organized as follows: Section II reviews MAC-based in-memory operations and related work. Section III details the proposed in-memory verification framework, covering both deterministic CNF-based and sampling-based methods. Section IV presents the experimental results, and Section V concludes the paper.

II. Background

A. MAC Operation

RRAM crossbar arrays inherently support analog MAC operations by storing weights as device conductances and applying inputs as row voltages. The resulting column currents naturally represent weighted sums of the applied inputs and are computed in parallel within a single evaluation cycle. This property enables highly efficient in-memory computation with massive parallelism and forms the foundation of many IMC architectures.

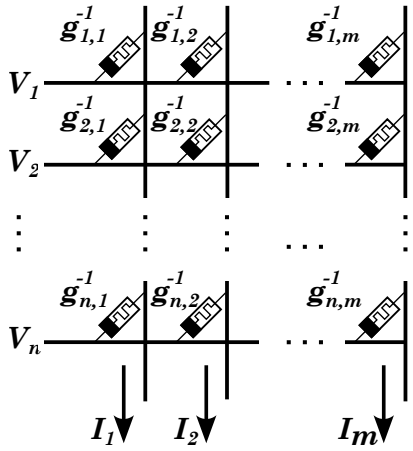


Fig. 1: MAC computation in RRAM crossbar

The RRAM devices within the crossbar structure, shown in Fig. 1, are programmed to conductance values $g_{j,k}$, or equivalently resistive values $r_{j,k} = g_{j,k}^{-1}$. When the input voltage vector $V_{in} = (V_1, \ldots, V_n)$ is applied simultaneously to the crossbar rows, the contributions of all activated cells accumulate along each column. As a result, up to m MAC operations can be performed in parallel across the m columns of the crossbar.

According to Kirchhoff's Current Law (KCL), the current measured at column j is given by

$$I_j = \sum_{k=1}^{n} V_k g_{j,k} \tag{1}$$

where V_k denotes the voltage applied to row k and $g_{j,k}$ denotes the conductance of the corresponding RRAM device. Thus, the output current of each column represents the accumulation of contributions from all participating rows.

The MAC operation can be compactly expressed in matrix form as

$$I = V_{in} G \tag{2}$$

where $I = (I_1, \ldots, I_m)$ denotes the vector of output currents, $V_{in} = (V_1, \ldots, V_n)$ is the input voltage vector, and the conductance matrix G is defined as

$$G = \begin{bmatrix} g_{1,1} & g_{1,2} & g_{1,3} & \cdots & g_{1,m} \\ g_{2,1} & g_{2,2} & \cdots & \cdots & g_{2,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ g_{n,1} & g_{n,2} & g_{n,3} & \cdots & g_{n,m} \end{bmatrix} \tag{3}$$

Consequently, the crossbar inherently performs m parallel MAC evaluations, each involving contributions from up to n rows, within a single evaluation cycle. While this capability has been widely exploited for matrix–vector multiplication and neuromorphic computing applications, the same current-accumulation mechanism can also be leveraged for Boolean computation.

In this work, Boolean variables and clauses are mapped onto crossbar rows and columns, respectively. The resulting MAC operations are interpreted in the Boolean domain to evaluate clause satisfaction and ultimately determine the satisfiability of the target CNF formula.

RRAM is selected as the target substrate because it naturally supports dense passive crossbar organization, non-volatile storage of programmed conductance states, and current-domain accumulation across columns. Other non-volatile memory technologies, such as Magnetic RAM (MRAM) and Phase-Change Memory (PCM), can also support forms of IM; however, differences in write mechanisms, resistance windows, endurance characteristics, and peripheral circuitry lead to different execution and sensing trade-offs [1].

B. Related Work

In-memory SAT solvers based on RRAM crossbar architectures have been proposed to enable self-verification of memristive systems by executing function clauses and sub-clauses directly within the memory array [7]. The approach exploits the native MAC capability of RRAM crossbars and evaluates Boolean functions through a hierarchical realization of sub-clauses and clauses inside the crossbar. While effective for in-memory SAT computation, the mapping is tightly coupled to the structure of the target Boolean function and requires intermediate clause evaluations and storage of internal results. Consequently, the computation remains sequential across logic levels and relies on exhaustive evaluation of input assignments to obtain exact SAT decisions, limiting scalability for large designs.

SAT-based equivalence checking for logic-in-memory architectures has also been investigated [8], [9]. In [9], functional equivalence between Majority-Inverter Graphs (MIGs) and RRAM-oriented HDL programs is established through SAT/SMT-based proofs of purity and equivalence at the behavioral level. The approach verifies that generated in-memory programs preserve the functionality of the original design, but relies on conventional off-chip solving engines. Similarly, [8] introduces an automated verification framework that translates majority-based in-memory implementations into SAT instances using a RRAM Sequence Graph (ReSG) representation and verifies them against a golden specification using SAT or SMT solvers. These methods provide exact verification guarantees, but the verification process itself remains external to the memory architecture.

Despite their effectiveness, existing approaches either require exhaustive SAT evaluation or rely on off-chip solvers, limiting scalability and preventing full exploitation of in-memory parallelism. In contrast, this work introduces an in-memory SAT evaluation method as a deterministic baseline, along with a sampling-based in-memory falsification method that executes randomized evaluation directly on RRAM crossbars through MAC-compatible operations. The proposed sampling approach trades completeness for scalability and enables rapid detection of functional discrepancies without requiring exhaustive exploration of the input space.

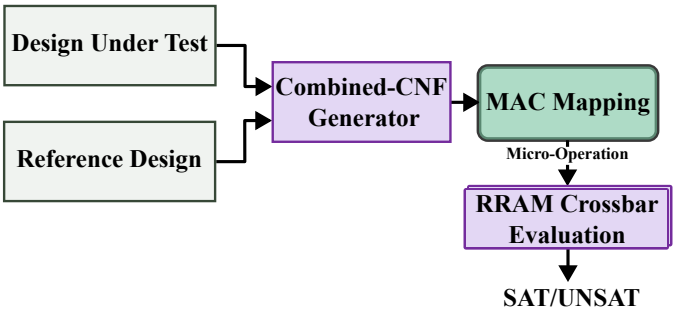


Fig. 2: Overall framework of the proposed CNF-based in-memory equivalence checking.

III. Proposed In-Memory Verification Framework

This section presents the proposed in-memory verification framework, which integrates Boolean reasoning directly into RRAM crossbar arrays. The framework supports both exact and sampling-based evaluation modes, enabling a flexible trade-off between completeness and computational cost. We assume an ideal binary RRAM crossbar with distinguishable high- and low-conductance states and threshold-based column sensing. Reported cycle counts include crossbar initialization and MAC evaluations, but exclude device variation, sneak-path currents, wire resistance, ADC resolution, and peripheral-control overhead. Consequently, the reported results should be interpreted as architectural-level rather than device-level estimates. While these non-idealities may affect absolute latency, energy consumption, and sensing margins, they do not alter the relative scaling behavior of exhaustive evaluation and sampling-based falsification considered in this work.

A. Method I: Deterministic CNF-Based In-Memory SAT Evaluation

Over the past decades, software-based SAT solvers have become a cornerstone of verification, particularly for the verification of logic equivalence and the validation of complex digital circuits. However, verification remains off-chip, failing to leverage the inherent parallelism of LiM architectures.

To address this gap, we introduce a deterministic in-memory SAT evaluation method that maps a Boolean formula in CNF directly onto an RRAM crossbar. The proposed approach enables exact SAT/UNSAT decisions to be computed entirely within the memory array, serving as a correctness-preserving baseline for in-memory verification.

The proposed in-memory SAT evaluation flow targets the equivalence checking of two arbitrary Boolean functions implemented as a Reference Design (RD) and a Design Under Test (DUT). The overall architecture is illustrated in Fig. 2. Given two Verilog descriptions, both designs are first processed using the ABC tool to generate intermediate Boolean representations. These representations are then translated into a combined CNF in DIMACS format, which encodes both functional logic and equivalence constraints.

To verify equivalence, the combined CNF includes a miter construction, where the corresponding outputs of RD and DUT

are compared using XOR relations. A satisfiable CNF instance indicates the existence of an input pattern that distinguishes the two designs, whereas unsatisfiability implies functional equivalence. DIMACS is employed due to its canonical structure and compatibility with clause-level evaluation, where each clause is represented as a disjunction of literals and the full formula as a conjunction of clauses. This formulation allows direct realization on RRAM crossbar arrays.

1) CNF Mapping onto the Crossbar: The CNF representation is inherently well-suited for execution on RRAM crossbar architectures. Clause-level disjunctions (OR operations) and their global conjunction (AND operation) can be efficiently realized using MAC operations, which are natively supported by crossbar arrays.

Let a CNF formula consist of M clauses defined over N variables. Each clause represents an OR operation over a subset of literals, while the full CNF evaluates to the AND of all clause results. In the proposed mapping, each clause is assigned to a dedicated crossbar column, enabling all clause evaluations to be performed in parallel. Each Boolean variable and its negation are mapped to separate crossbar rows, yielding $2N$ rows in total and eliminating the need for inverter circuitry. This explicit representation allows both positive and negated literals appearing in the CNF formula to be evaluated directly within the crossbar.

To enable MAC-based execution, the CNF formula is translated into a set of crossbar micro-operations following the MAC mapping methodology described in [10]. Each micro-operation specifies the clause column to be evaluated together with the rows corresponding to the literals participating in that clause. These micro-operations define the row-column connectivity of the crossbar and are generated directly from the CNF representation.

2) Crossbar Structure and Initialization Method: During crossbar initialization, the logical structure of the CNF formula is encoded into the resistive states of the RRAM devices. Specifically, the presence or absence of a literal in a clause is represented by programming the corresponding device at the intersection of the literal row and clause column. Initialization is performed using a sequential clause-wise programming strategy. In each initialization cycle, one clause column is configured. For that clause, devices located on rows corresponding to its literals are programmed to the conductive state, while the remaining devices in the column remain non-conductive. This procedure is repeated for all M clause columns, resulting in M initialization cycles. After initialization, the complete CNF structure is embedded in the crossbar as a pattern of resistive states. Once initialized, a single MAC cycle is performed to evaluate all clauses in parallel. For a given input assignment, the values of all variables are applied simultaneously to the corresponding rows, and each clause column produces an output current corresponding to its associated clause. The column currents are then read to determine the satisfaction status of the individual clauses, and the resulting clause-status bits are stored in an output register. Since the CNF formula is satisfied only when all clauses evaluate to logic one, the SAT decision is obtained directly from the contents of this register.

Thus, the initialization phase is performed only once to program the CNF structure into the crossbar. After initialization, each input assignment can be evaluated using a single MAC operation. Therefore,

$$\#\text{Initialization Cycles} = M, \tag{4}$$

and the per-assignment evaluation cost is

$$\#\text{MAC Cycles}_{\text{per input assignment}} = 1. \tag{5}$$

The resulting crossbar configuration consists of $2N$ rows and M columns, where N and M denote the numbers of CNF variables and clauses, respectively. Thus, the required crossbar size is

$$\#\text{Crossbar } (r \times c) = 2N \times M. \tag{6}$$

3) SAT Decision and Equivalence Interpretation: Following the MAC evaluation, the clause-status bits stored in the output register determine the satisfiability of the CNF formula for the current input assignment. Since a CNF formula is satisfied only when all of its clauses evaluate to logic one, the formula is satisfiable if all clause-status bits are equal to one. Conversely, the presence of at least one zero-valued clause-status bit indicates that the CNF formula is unsatisfied for the current assignment.

For equivalence checking, the CNF formulation incorporates the miter constraints between the Reference Design (RD) and the Design Under Test (DUT). Consequently, a satisfiable CNF instance indicates the existence of a distinguishing input assignment and, therefore, proves functional non-equivalence. In contrast, if no satisfying assignment exists, the CNF is unsatisfiable, implying functional equivalence between RD and DUT.

It is important to note that a single crossbar evaluation corresponds to evaluating the CNF formula for one specific input assignment. Therefore, for a CNF defined on the N variables, a complete and deterministic SAT evaluation requires examining all possible input assignments 2^N. While this exhaustive process provides exact SAT/UNSAT decisions, its exponential complexity motivates the sampling-based falsification approach presented in the following subsection.

Listing 1 illustrates the generated micro-operation description for a small CNF instance. First, each variable and its complement are mapped to separate crossbar rows. For example, variable `2` and its complement `-2` are assigned to rows 0 and 1 (i.e., `R0` and `R1`), respectively. After this row mapping, each clause is assigned to a dedicated crossbar column, enabling column-wise MAC evaluation.

The micro-operation entries specify the row–column connectivity used to realize the CNF formula within the crossbar. For example, the first clause $(2 \vee \neg 3 \vee \neg 4)$ is mapped to Column 0 and represented by the micro-operation :

```
0 False 0 True 3 True 5 True
```

Listing 1: Example CNF and corresponding MAC-based micro-operations

```
# CNF input
p cnf 5 5
2 -3 -4 0
-2 3 0
-2 4 0
-5 0
2 0
-----------------------------------------------------

# Literal-to-Row Mapping
2/R0  -2/R1
3/R2  -3/R3
4/R4  -4/R5
5/R6  -5/R7

# MICRO-OPERATIONS (Clause Evaluation)
0 False 0 True 3 True 5 True
1 False 1 True 2 True
2 False 1 True 4 True
3 False 7 True
4 False 0 True
```

Here, `0 False` indicates that Column 0 is selected and connected to ground, while `0 True`, `3 True`, and `5 True` activate rows 0, 3, and 5, corresponding to literals 2, $\neg 3$, and $\neg 4$, respectively. The contributions of the activated rows are accumulated on the selected column through the MAC operation, and the resulting column current is then read to determine the satisfaction status of the clause.

B. Method II: Sampling-Based In-Memory Falsification

The deterministic CNF-based method described in the previous subsection provides exact SAT/UNSAT decisions but requires exhaustive evaluation over all 2^N input assignments, which is computationally infeasible for large designs. To address this limitation, we introduce a sampling-based in-memory falsification method that leverages randomized input exploration to efficiently detect functional mismatches between designs.

Unlike formal equivalence checking approaches that aim to prove equivalence exhaustively, the proposed method focuses on *falsification*, i.e., identifying input patterns that expose discrepancies between RD and DUT. This distinction is particularly relevant in practical verification flows, where rapid detection of design bugs is often prioritized during early validation stages, complementing complete formal guarantees [11]–[13].

1) Miter-Based Formulation: Given an RD and a DUT with shared primary inputs $x \in \{0,1\}^N$, equivalence checking is reduced to evaluating a single-output miter circuit defined as:

$$D(x) = \bigvee_{j=1}^{O} \left(y_j^{RD}(x) \oplus y_j^{DUT}(x) \right),$$

where $D(x) = 1$ indicates a functional mismatch for input x.

Thus, the existence of an input x such that $D(x) = 1$ implies non-equivalence between the two designs.

2) Sampling-Based Evaluation: The input space is explored through randomized sampling, following a Monte Carlo evaluation paradigm. At each iteration k, an input vector is drawn uniformly at random:

$$x^{(k)} \sim \mathcal{U}(\{0,1\}^N),$$

and the corresponding miter output $D(x^{(k)})$ is evaluated directly on the RRAM crossbar.

Each sampled input corresponds to a complete in-memory execution of the mapped AIG representation. The process continues until a mismatch is detected or a predefined sampling budget K is reached.

3) Decision Rule and One-Sided Guarantee: The decision procedure follows a one-sided error paradigm:

- If there exists at least one sampled input $x^{(k)}$ such that $D(x^{(k)}) = 1$, the method immediately returns Non-equivalent. This result is exact, as the sampled input serves as a concrete counterexample that demonstrates a functional discrepancy.
- If no mismatch is observed after the K samples, the method returns *Inconclusive*, indicating that no discrepancy was detected within the explored input space.

Importantly, the method does not produce false positives: any detected mismatch corresponds to a valid functional difference. However, the absence of mismatches over a finite number of samples does not constitute proof of equivalence.

4) Detection Probability and Confidence: Let

$$p = \Pr_{x \sim \mathcal{U}(\{0,1\}^N)}[D(x) = 1]$$

denote the (unknown) probability that a random input reveals a mismatch.

In practice, p is unknown and may be very small, which limits worst-case detection guarantees. To address this, we define a *target detectable mismatch rate* $p_0 > 0$, representing the minimum mismatch probability of interest. For a given sampling budget K, the probability of failing to detect any mismatch when $p \geq p_0$ is bounded by:

$$\Pr[\text{miss all mismatches}] \leq (1 - p_0)^K.$$

Equivalently, the probability of detecting at least one mismatch is:

$$\Pr[\text{detect mismatch}] \geq 1 - (1 - p_0)^K.$$

This formulation provides a practical interpretation of confidence: for any chosen pair (K, p_0), the designer can quantify the probability of detecting bugs that occur with probability at least p_0.

Figure 3 illustrates that frequently occurring mismatches are detected with only a small number of samples, whereas rarer mismatches require a larger sampling budget. For example, when $p_0 = 10^{-3}$, approximately 3,000 samples provide about 95% detection probability.

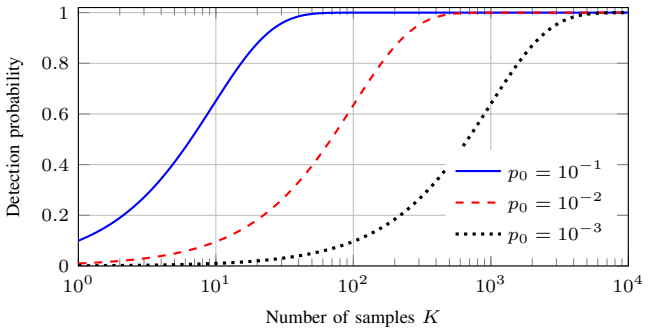


Fig. 3: Probability of detecting at least one mismatch after K independent samples for different target mismatch rates p_0.

5) Execution Model and Cycle Accounting: Each sampled input corresponds to one full crossbar evaluation of the AIG-based miter on the crossbar. The execution follows a MAC-based model in which RRAM devices are initialized once and reused across multiple evaluations.

Specifically, a one-time initialization phase of N_{AND} cycles is performed, with one cycle used to configure each AND-node column according to the AIG connectivity. This initialization is independent of the number of samples and is reused across all sampled evaluations. After initialization, computation is controlled through the voltages applied to the crossbar rows. Each AND node is evaluated by one MAC operation; because dependent nodes are processed in topological order, each sampled input requires N_{AND} MAC cycles.

Let N_{AND} denote the number of AND nodes in the AIG. The one-time initialization cost is N_{AND} cycles, while each sampling iteration requires N_{AND} MAC cycles. The total computational cost for K samples is given by:

$$\#\text{Total Cycles} = (1 + K) \cdot N_{\text{AND}} \tag{7}$$

If a mismatch is detected during sampling, the process terminates early, resulting in significantly reduced execution time in practice.

In the AIG representation, the inputs to AND gates consist of either primary inputs or the outputs of other AND gates. Each distinct input appearing at the inputs of AND gates is treated as a Boolean variable that may be reused across multiple nodes, with negation encoded on the connecting edges. As in Method I, each such variable and its negation are explicitly represented using two crossbar rows.

Let N_{var} denote the number of distinct variables appearing at the inputs of AND gates. The resulting crossbar realization therefore requires $2N_{\text{var}}$ rows. Since each AND node is assigned to a separate crossbar column, the resulting crossbar configuration has dimensions $2N_{\text{var}} \times N_{\text{AND}}$. Note that N_{var} is generally smaller than $2N_{\text{AND}}$, since variables can be reused between multiple AND nodes. The following example illustrates the complete Method II flow, including AIG mapping, sample generation, stopping conditions, and cycle accounting. Consider a miter whose output function is

$$D(a, b, c) = (a \wedge b) \vee c, \tag{8}$$

where $D = 1$ indicates a functional mismatch between the reference design and the design under test.

Using complemented AIG edges, the OR operation is represented as

$$(n_1 = a \wedge b), \tag{9}$$

$$(n_2 = \neg n_1 \wedge \neg c), \tag{10}$$

$$D = \neg n_2. \tag{11}$$

Thus, the AIG contains two dependent AND nodes. Node n_1 is mapped to column C_0, while n_2 is mapped to column C_1. Since n_2 depends on n_1, the nodes are evaluated in topological order. Under the adopted architectural model, each AND-node evaluation contributes one MAC cycle. Consequently, each sampled input assignment requires two MAC cycles.

Listing 2: Example AIG mapping and sampling-based falsification flow

```
# Example Miter Output Function
D(a,b,c) = (a AND b) OR c
n1 = AND(a,b)
n2 = AND(!n1,!c)
D  = !n2
-----------------------------------------------------

# Literal-to-Row Mapping
a/R0   -a/R1
b/R2   -b/R3
n1/R4  -n1/R5
c/R6   -c/R7

# Column mapping
n1/C0 : (R0,C0) (R2,C0)
n2/C1 : (R5,C1) (R7,C1)

N_AND=2
N_var=4
Crossbar=8x2

# Sampling parameters
p0=0.25
delta=0.1
K=9

# One-time Initialization
Configure crossbar cells for C0 and C1

# Case A: counterexample found
k=1 : 000 -> n1=0 -> n2=1 -> D=0
k=2 : 100 -> n1=0 -> n2=1 -> D=0
k=3 : 110 -> n1=1 -> n2=0 -> D=1

STOP
Result -> NON-EQUIVALENT
Counterexample=110

# Case B: budget exhausted
k=1,...,9 : D=0

STOP
Result -> INCONCLUSIVE
```

The distinct signals appearing as inputs to AND nodes are a, b, n_1, and c, yielding $N_{\text{var}} = 4$. Their positive and complemented forms require $2N_{\text{var}} = 8$ rows. Since the AIG contains $N_{\text{AND}} = 2$ AND nodes, two columns are required. Therefore, the crossbar dimensions are

$$(2N_{\text{var}}) \times N_{\text{AND}} = 8 \times 2. \tag{12}$$

For illustration, assume a target detectable mismatch rate of $p_0 = 0.25$ and an acceptable miss probability of $\delta = 0.1$. The required sampling budget is

$$K = \left\lceil \frac{\log(\delta)}{\log(1 - p_0)} \right\rceil = 9. \tag{13}$$

The corresponding probability of detecting at least one mismatch with occurrence probability $p \geq p_0$ is bounded by

$$\Pr[\text{detect}] \geq 1 - (1 - p_0)^K = 1 - 0.75^9 \approx 0.925. \tag{14}$$

Therefore, at most nine random input assignments are evaluated. The value of K limits only the number of sampled inputs; it does not truncate or otherwise alter the mapped AIG.

In Case A, the search terminates immediately when the third sampled assignment activates the miter output. Since $D = 1$, the assignment $abc = 110$ is an exact witness of non-equivalence. Each sample requires $N_{\text{AND}} = 2$ MAC cycles. Therefore, three samples require six MAC cycles. Including the two initialization cycles, the total cost is

$$2 + 3 \times 2 = 8 \text{ cycles}. \tag{15}$$

In Case B, no sampled assignment activates the miter output. The procedure continues until the budget $K = 9$ is exhausted and returns *Inconclusive*, not *Equivalent*. Under the architectural model, the worst-case cost is

$$2 + 9 \times 2 = 20 \text{ cycles}. \tag{16}$$

IV. Experimental evaluation

This section evaluates the proposed in-memory verification framework on standard combinational benchmark suites, focusing on latency and non-equivalence detection. We examine two strategies: the deterministic CNF-based method and the sampling-based falsification method tailored to in-memory computing.

A. Benchmark Suite and Setup

We evaluate the proposed approaches using benchmarks from the ISCAS-85 and IWLS-2005 suites [14], [15], which include a diverse set of arithmetic and control-oriented circuits with varying structural complexity.

To evaluate the non-equivalence detection capability of the proposed methods, benchmark pairs were constructed from the original circuits. For each benchmark, the original circuit was used as the reference design (RD), while a modified version served as the design under test (DUT). The DUTs were generated by manually introducing functional modifications at randomly selected logic gates, including gate-type substitutions and the insertion of additional inversions. These modifications create functional discrepancies between the RD and DUT while preserving the overall circuit structure.

For each benchmark pair, a miter circuit was generated using Yosys. The resulting miter output evaluates to logic one whenever a functional mismatch exists between the RD and DUT. Counterexamples were obtained by solving the generated miter instances with the Z3 SMT solver and were used only for validation. Since equivalence checking is performed on the miter output, a detected counterexample corresponds to at least one differing primary output between the reference and modified designs. The number of distinct primary outputs was not constrained during fault insertion and depends on the propagation effects of the modified logic gate in each benchmark.

All reported results focus on cycle-level performance metrics, including initialization cycles, MAC cycles, total execution cycles, and the required crossbar dimensions. These metrics reflect the architectural latency and scalability of the proposed methods.

B. CNF-Based Exhaustive Equivalence Checking

Table I reports the results of the CNF-based equivalence checking method. In this approach, the Boolean function is translated into CNF and evaluated exhaustively over all possible input assignments.

The initialization cost is a one-time overhead required to program the CNF structure into the crossbar. Once programmed, all input assignments are evaluated using repeated MAC operations without reprogramming the array. Table I reports the one-time initialization cost together with the MAC cost required to evaluate a single input assignment. Since this method requires exhaustive enumeration of all possible assignments, the total cycle count grows exponentially with the number of Boolean variables and is therefore given by

$$\#\text{Total Cycles} = M + 2^N \tag{17}$$

The table further summarizes the benchmark characteristics (#PI, #PO, clauses, and variables), together with the initialization cost, MAC cycles per input assignment, total cycles for exhaustive evaluation, and the required crossbar dimensions.

As shown in Table I, while the CNF-based method provides deterministic and complete verification guarantees, its latency becomes prohibitive for large benchmarks due to the exponential growth in the number of evaluated input assignments. This behavior highlights the scalability limitations of exhaustive verification for complex designs.

C. Sampling-Based In-Memory Falsification

Table II reports the results of the proposed sampling-based in-memory falsification method. Unlike the exhaustive CNF-based approach, the proposed method evaluates randomly sampled input assignments directly on the crossbar using a Monte Carlo strategy. Each sampled assignment corresponds to one execution of the mapped AIG-based miter. Non-equivalence is detected as soon as a sampled input activates the miter output, yielding a valid counterexample. If no mismatch is observed

TABLE I: Evaluation results of the CNF-based in-memory equivalence checking method

	Benchmark			Proposed Method I					
	Name	#PI	#PO	Clauses	Variables	Init. Cycle	MAC Cycle*	Total Cycle	Crossbar($r \times c$)
ISCAS-85	c17	5	2	72	36	72	1	$72 + 2^{36}$	72×72
	c432	36	7	926	581	926	1	$926 + 2^{581}$	1162×926
	c499	41	32	2346	1822	2346	1	$2346 + 2^{1822}$	3644×2346
	c1355	41	32	1636	460	1636	1	$1636 + 2^{460}$	920×1636
	c2670	233	140	7214	3148	7214	1	$7214 + 2^{3148}$	6296×7214
	c3540	50	22	2859	3718	2859	1	$2859 + 2^{3718}$	7436×2859
	c5315	178	123	6826	6026	6826	1	$6826 + 2^{6026}$	12052×6826
IWLS-2005	9sym_d	9	1	232	224	232	1	$232 + 2^{224}$	448×232
	con1f1	7	2	93	198	93	1	$93 + 2^{198}$	396×93
	exam1_d	3	1	35	77	35	1	$35 + 2^{77}$	154×35
	max46_d	9	1	572	836	572	1	$572 + 2^{836}$	1672×572
	newill_d	8	1	96	79	96	1	$96 + 2^{79}$	158×96
	newtag_d	8	1	57	48	57	1	$54 + 2^{48}$	96×57
	rd53f1	5	3	187	168	187	1	$187 + 2^{168}$	336×187
	rd73f1	7	3	432	437	432	1	$432 + 2^{437}$	874×432
	rd84f1	8	1	312	322	312	1	$312 + 2^{322}$	644×312
	sao2f1	10	1	153	183	153	1	$153 + 2^{183}$	366×153
	sym10_d	10	1	264	300	264	1	$264 + 2^{300}$	600×264

* MAC cycles are reported per evaluated input assignment.

TABLE II: Evaluation results of the sampling-based in-memory non-equivalence detection method over 100 random seeds with sampling budget $K = 10{,}000$

	Benchmark			Proposed Method II						
	Name	#PI	#PO	Samples avg.±std.	Range	$\hat{p}$	N_{var}	Init.	MAC avg.	Total avg.
ISCAS-85	c17	5	2	1.86 ± 1.22	1–9	0.590	9	5	9.3	14.3
	c432	36	7	2.29 ± 1.86	1–11	0.468	208	173	396.2	569.2
	c499	41	32	4.19 ± 3.61	1–15	0.253	479	439	1839.4	2278.4
	c1355	41	32	8.99 ± 7.68	1–35	0.126	528	488	4387.1	4875.1
	c2670	233	140	1.07 ± 0.26	1–2	0.963	488	362	387.3	749.3
	c3540	50	22	1.01 ± 0.10	1–2	0.969	923	874	882.7	1756.7
	c5315	178	123	2.19 ± 1.86	1–15	0.450	216	176	385.4	561.4
IWLS-2005	9sym_d	9	1	1.23 ± 0.55	1–4	0.830	75	67	82.4	149.4
	con1f1	7	2	4.52 ± 3.83	1–16	0.193	8	5	22.6	27.6
	exam1_d	3	1	1.01 ± 0.20	1–4	0.891	12	9	9.09	18.09
	max46_d	9	1	9.74 ± 7.26	1–50	0.123	146	138	1344.1	1482.1
	newill_d	8	1	16.06 ± 14.36	1–47	0.079	37	30	481.8	511.8
	newtag_d	8	1	2.67 ± 1.91	1–11	0.343	25	18	48.1	66.1
	rd53f1	5	3	4.39 ± 3.91	1–17	0.215	20	16	70.2	86.2
	rd73f1	7	3	1.47 ± 0.82	1–4	0.628	35	29	42.6	71.6
	rd84f1	8	1	2.96 ± 2.17	1–14	0.355	102	95	281.2	376.2
	sao2f1	10	1	14.82 ± 9.94	1–42	0.094	51	42	622.44	664.44
	sym10_d	10	1	8.14 ± 6.72	1–27	0.123	96	87	708.2	795.2

within the sampling budget, the result remains inconclusive, since finite sampling cannot guarantee equivalence.

To characterize the statistical behavior of the method, each benchmark was evaluated using 100 independent random seeds with a sampling budget of $K = 10,000$. Table II summarizes the resulting detection statistics. The columns *#PI* and *#PO* denote the numbers of primary inputs and outputs, respectively. *Samples avg.*$(\pm)$*std.* reports the mean and standard deviation of the number of sampled evaluations required to detect the first counterexample, while *Range* gives the corresponding minimum and maximum values. The column $(\hat{p})$ denotes the empirical mismatch probability estimated from 10,000 randomly generated input assignments. The parameter (N_{var}) represents the number of distinct signals appearing at the inputs of the mapped AIG AND nodes.The *Init.*, *MAC avg.*, and *Total avg.* columns report the initialization cost, the average MAC cycles required until mismatch detection, and the corresponding total cycle count, respectively.

The results show that non-equivalence is typically detected after only a small number of sampled evaluations. For many benchmarks, the average number of required samples is close to one, indicating that mismatches are encountered almost immediately. Even for benchmarks with lower mismatch probabilities, the required sample counts remain several orders of magnitude smaller than the exhaustive exploration of the complete input space required by Method I. These observations demonstrate the effectiveness of randomized falsification for rapidly identifying functional discrepancies in large verifica-

TABLE III: Representative mismatch-detection results from a single execution of the proposed sampling-based falsification method

	Benchmark	Representative Detection Run				
	Name	#Samples	Init. Cycle	MAC Cycle	Total Cycle	Crossbar ($r \times c$)
ISCAS-85	c17	1	5	5	10	18×5
	c432	2	173	346	519	416×173
	c499	4	439	1756	2195	958×439
	c1355	12	488	5856	6344	1056×488
	c2670	1	362	362	724	976×362
	c3540	1	874	874	1748	1846×874
	c5315	2	176	352	528	432×176
IWLS-2005	9sym_d	1	67	67	134	150×67
	con1f1	1	5	5	10	16×5
	exam1_d	3	9	27	36	24×9
	max46_d	6	138	828	966	292×138
	newill_d	29	30	870	900	74×30
	newtag_d	1	18	18	36	50×18
	rd53f1	1	16	16	32	40×16
	rd73f1	2	29	58	87	70×29
	rd84f1	4	95	380	475	204×95
	sao2f1	23	42	966	1008	102×42
	sym10_d	2	87	174	261	192×87

tion instances.

To complement the statistical results of Table II, Table III reports representative mismatch-detection results obtained from a single execution of the proposed method for each benchmark. While Table II characterizes average behavior over multiple random seeds, Table III illustrates the actual number of sampled evaluations and execution cycles required in a representative run. The table reports the number of samples required to detect a mismatch together with the corresponding initialization cost, MAC cycles, total execution cycles, and crossbar dimensions.

As shown in Table III, non-equivalence is frequently detected after only one or two sampled evaluations. Consequently, the total cycle counts remain very small compared with the exhaustive verification costs reported in Table I. While Method I guarantees complete verification by evaluating the entire input space, the proposed sampling-based approach often terminates immediately after observing a counterexample. This behavior substantially reduces the required execution time while preserving exact correctness whenever a mismatch is detected.

D. Quantitative Comparison and Scalability Analysis

Tables I, II, and III enable a quantitative comparison between the exhaustive CNF-based method and the proposed sampling-based falsification approach in terms of latency, detection capability, and scalability.

a) Latency Reduction: The CNF-based method requires exhaustive evaluation over all possible input assignments, resulting in a total cycle count that grows exponentially with the number of Boolean variables. In contrast, the sampling-based falsification method evaluates only a limited number of sampled inputs and therefore scales linearly with the number of samples. For large benchmarks, this leads to a substantial reduction in execution latency. For example, for benchmark `c432`, the CNF-based method requires $926+2^{581}$ total cycles, whereas the proposed method detects non-equivalence with an average of 569.2 cycles over 100 random seeds. This illustrates the impracticality of exhaustive evaluation for realistic designs.

More generally, the sampling-based method reduces the effective latency from exponential complexity to a sample-dependent linear cost. In practice, mismatches are typically detected within a small number of samples, enabling scalable falsification of non-equivalent designs that are otherwise intractable using CNF-based execution.

b) Crossbar Utilization: The crossbar dimensions required by the sampling-based method are also substantially smaller. Since only variables appearing at the inputs of AND nodes need to be represented, the number of rows is significantly reduced compared to the CNF-based approach, which must explicitly encode all clauses and literals. This reduction directly translates into improved area efficiency and lower initialization overhead.

c) Detection Capability: The CNF-based method provides deterministic and complete verification guarantees, as it exhaustively evaluates all possible input assignments. The proposed falsification method, on the other hand, deterministically detects non-equivalence once a counterexample is observed, but does not attempt to prove equivalence under finite sampling.

However, the probability of missing a mismatch decreases exponentially with the number of samples. In practice, as shown in Table II, non-equivalence is detected in all 100 runs for every reported benchmark, with no inconclusive outcomes under the selected sampling budget.

d) Overall Scalability: In summary, the CNF-based method serves as a deterministic reference but suffers from exponential latency and limited scalability. The proposed

sampling-based falsification approach achieves significant reductions in latency and crossbar utilization, while effectively detecting non-equivalent designs. This trade-off makes the method particularly well-suited for large-scale in-memory computing architectures.

V. Conclusion

This paper presented an in-memory verification framework for RRAM crossbar architectures based on MAC-style computation. Two complementary strategies were investigated. The first maps CNF formulas directly onto RRAM crossbars and performs deterministic SAT evaluation, providing exact verification guarantees through exhaustive exploration of the input space. The second introduces a sampling-based falsification approach that exploits randomized input exploration to efficiently detect functional mismatches without requiring complete enumeration of all assignments.

Experimental results on standard benchmarks demonstrated the trade-off between completeness and scalability. While the deterministic CNF-based approach provides exact verification results, its execution cost grows exponentially with the number of Boolean variables. In contrast, the proposed sampling-based method detects non-equivalence after only a small number of sampled evaluations in most cases, resulting in substantially lower execution latency and reduced crossbar requirements.

These results indicate that sampling-based in-memory falsification can serve as an effective pre-verification strategy for rapidly identifying design errors, while deterministic in-memory SAT evaluation remains available when complete verification guarantees are required. Future work includes extending the framework to sequential circuits, investigating adaptive and guided sampling strategies, and incorporating realistic RRAM device non-idealities and sensing constraints into the evaluation flow.

Acknowledgments

This work was supported by the German Research Foundation (DFG) with project PLiM (DR 287/35-1, DR 287/35-2, and SH 1917/1-2), and by the German Ministry for Research, Technology, and Space (BMFTR) with project ExaVerse (grant number 01IW25003).

References

[1] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, vol. 15, no. 7, pp. 529–544, Jul 2020.

[2] K. Sun, J. Chen, and X. Yan, "The future of memristors: materials engineering and neural networks," *Advanced Functional Materials*, vol. 31, no. 8, p. 2006773, 2021.

[3] S. Frerix, S. Shirinzadeh, S. Fröhlich, and R. Drechsler, "ComPRIMe: A compiler for parallel and scalable ReRAM-based in-memory computing," in *Proceedings of the 2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'19)*. Bremen, Germany: IEEE, 2019, pp. 1–6.

[4] F. Shirinzadeh, A. Kole, K. Datta, S. Shirinzadeh, and R. Drechsler, "A comprehensive synthesis and verification approach for RRAM-based neuromorphic computing," in *Proceedings of the Euromicro Conference Series on Digital System Design (DSD)*, Italy, 2025.

[5] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (PLiM) computer," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 427–432.

[6] S. Froehlich and R. Drechsler, "LiM-HDL: HDL-based synthesis for in-memory computing," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 1395–1400.

[7] F. Shirinzadeh, A. Deb, S. Shirinzadeh, A. Kole, K. Datta, and R. Drechsler, "In-memory SAT-Solver for self-verification of programmable memristive architectures," in *37th International Conference on VLSI Design and 23rd International Conference on Embedded Systems, VLSID 2024, Kolkata, India, January 6-10, 2024*. IEEE, 2024, pp. 384–389.

[8] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler, "Automated equivalence checking method for Majority-based in-memory computing on ReRAM crossbars," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2023, p. 19–25.

[9] S. Froehlich and R. Drechsler, "Generation of verified programs for in-memory computing," in *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 815–820.

[10] F. Shirinzadeh, K. Datta, A. Kole, S. Shirinzadeh, and R. Drechsler, "An HDL-centric synthesis and verification approach for RRAM-based neuromorphic systems," *Microprocessors and Microsystems*, vol. 122, p. 105278, 2026.

[11] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, 1991, pp. 90–99.

[12] R. M. Karp and M. Luby, "Monte-carlo algorithms for enumeration and reliability problems," in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, 1983, pp. 56–64.

[13] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *Principles and Practice of Constraint Programming*. Springer Berlin Heidelberg, 2013, pp. 200–216.

[14] M. Hansen, H. Yalcin, and J. Hayes, "Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering," *IEEE Design Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[15] C. Albrecht, "IWLS 2005 benchmarks," International Workshop on Logic Synthesis (IWLS), 2005, available: http://iwls.org/iwls2005/benchmarks.html.