Data Flow Obfuscation: A new Paradigm for Obfuscating Circuits

Kimia Zamiri Azar*, Hadi Mardani Kamali†, Shervin Roshanisefat*, Houman Homayoun‡, Christous P Sotiriou†, and Avesta Sasan*

Abstract—In this paper, unlike almost all state-of-the-art obfuscation solutions that focus on the functional/logic obfuscation, we introduce a new paradigm, called data flow obfuscation, which exploits the essence of asynchronicity. In data flow obfuscation, by benefiting from the handshaking mechanism of asynchronous circuits, the system’s FFs/latches will operate out of sync. Hence, the adversary has no sufficient knowledge to apply unrolling/BMC. Also, due to the inherent asynchronicity, the exact time of writing/capturing data into/from the scan chain becomes hidden. Hence, the SAT attack cannot be applied even while scan chain access is open. Moreover, our new proposed paradigm creates stateful/oscillating combinational cycles into the design which extensively boosts the difficulty of modeling this technique. We also demonstrate how data flow obfuscation could easily be integrated with any circuit at low overhead while there is no limitation such as compromising test flow.

Index Terms—Logic Obfuscation, Desynchronization.

I. INTRODUCTION

The ever-increasing cost of integrated circuits (IC) manufacturing has forced many design houses to become fabless [1]. Outsourcing the stages of the manufacturing supply chain to the third-party facilities with no reliable monitoring on them results in emerging multiple forms of security threats such as IC overproduction, reverse engineering (RE), Intellectual Property (IP) theft, etc. [2, 3]. To combat these threats, logic obfuscation a.k.a. logic locking, as a proactive scheme adds post-manufacturing programming capability into the circuits [4, 5]. Logic obfuscation is the process of hiding the correct functionality of a circuit, during the stages at untrusted parties, when the programming value, referred to as the key, is unknown/incorrect. Only once the correct key is provided, the circuit behaves correctly, and the correct key would be initiated in its tamper-proof non-volatile memory (tpNVM) after fabrication via a trusted party.

Due to the importance of logic obfuscation, many studies have evaluated the effectiveness of this countermeasure [6, 7]. Amongst all state-of-the-art threats on logic obfuscation, the Boolean satisfiability (SAT) attack has seriously challenged the effectiveness of the vast majority of existing logic obfuscation solutions [8]. In the SAT attack, as an oracle-guided attack, the adversary has access to (1) one successfully reverse-engineered yet locked netlist†, and (2) the activated/functional IC with open access to the scan chain. By getting inspiration from the miter circuit used in formal verification, in the SAT attack, a SAT solver is employed to iteratively find a specific set of inputs (one per each iteration), called discriminating input patterns (DIPs) that eliminate all incorrect keys leading to recovering the correct functionality of the circuit.

The main strength of the SAT attack comes from two important factors: (1) The pruning power of each DIP that can rule out a high portion of incorrect keys; (2) Exploiting the scan chain access to apply the SAT attack on each combinational logic, separately. Hence, most studies on logic obfuscation could be categorized into the following categories:

(1) Some techniques try to weaken the pruning power of each DIP, helping exponentially increases the number of required SAT iterations [24–26]. In such techniques, the SAT attack can only rule out a few incorrect keys (the best case is one incorrect key) per each iteration. Hence, the SAT attack faces an exponential iterations w.r.t. the key size. However, these techniques suffer from various structural vulnerabilities that were eventually exploited to break them [10, 12, 38]. Besides, these techniques also suffer from very low output corruption. So, the adversary relies on approximate key with an extremely low error rate, which could be found by using approximate SAT attacks [11].

(2) Since SAT inputs are in Conjunctive Normal Form (CNF), a SAT attack works perfectly fine if the logic obfuscation is of Boolean nature. Hence, few recent studies lock the properties of the circuit, which cannot be translated to CNF, or it traps the SAT solver in an infinite loop, such as delay locking [30], or cyclic obfuscation [27–29, 39]. For instance, in delay locking, the delay of timing paths are obfuscated, which is not translatable to CNF. However, this breed of obfuscation techniques are already broken using newer attacks [13–15, 40].

(3) Some techniques significantly increase the runtime of each iteration of the SAT solver [31, 32]. In such techniques, by exploiting the strength of symmetric routing structures, such as permutation networks or crossbars, the complexity of the SAT circuit per each iteration will be increased significantly. However, the preliminary versions of routing-based obfuscation techniques [31, 32] are broken recently showing the vulnerability of this group as well [16, 17]. The state-of-the-art routing-based obfuscation technique, called InterLock [16] twists logic and routing obfuscation to still get benefits of the complexity brought by the symmetric

†Two scenarios: (1) Adversary as end-user purchases and delayers the activated chip; However, the content of tpNVM will be wiped-out during reverse engineering. (2) Adversary as staff at foundry before fabrication does have access to the layout, but the key is not provided to the foundry [9].
structures. However, it still incurs a slightly large area overhead (due to routing structures), which makes them less usable, particularly while the circuits are moderately small.

(4) Since the SAT attack requires access to the scan chain to target each combinational logic part separately, some techniques try to block/obfuscate the scan chain for restricting any unauthorized scan chain access [20, 36, 41, 42]. Such techniques show that with restricted scan access, engaging a traditional logic obfuscation, e.g. SLL [43], might guarantee the resiliency. However, these techniques are eventually broken using unrolling-based SAT attack and BMC [18, 19], called sequential SAT attack. Also, in some of these cases, it is assumed that the tester is a trusted party that should have the correct key; however, it is a hard assumption to maintain in many practical cases. The most recent study in this category, DisORC + TRLL [37], is a combination of true random logic locking with restricted scan access. Although none of the existing attacks could break this new countermeasure, it is still suffers from a few test-related restrictions. For instance, the functional test must be done at trusted entity, and accomplishing it using PI/PO might reduce the testability coverage depending on the topological hierarchy of the design-under-test, and also might increase the testability time/complexity substantially.

A. Motivation and Contributions

Table I provides a comprehensive comparison of state-of-the-art logic obfuscation techniques in all four aforementioned categories. As shown in Table I, a reliable logic obfuscation technique must be (1) resilient against both combinational and sequential SAT attack, (2) added without compromising the test flow. To fulfill these requirements, in this paper, we introduce a new logic obfuscation paradigm, called data flow obfuscation, whose main contributions are as follows:

(1) In data flow obfuscation, getting inspired by asynchronous circuits, the data flow is asynchronously key-controlled. Since the sequential SAT attack unrolls the FF-to-FF (flip-flop) paths per each iteration (each iteration resembles each clock cycle), in data flow obfuscation, a small portion of the FF would be replaced with latches controlled by asynchronous obfuscated controllers. Hence, the timing (flow) of the latches is unknown for the adversary (locked). So, the adversary has no sufficient knowledge to do the unrolling cycle-accurately, and the sequential SAT attack is no longer applicable to a data flow obfuscated circuit.

(2) With an obfuscated asynchronous controller, having access to the scan chain without any information about the exact timing (locked timing) of arrival/departure of data does not provide any advantages for the adversary. Hence, as shown in Table I, with open scan chain access, and without any limitation on the test phase, the data flow obfuscation is resilient against all state-of-the-art attacks at low overhead.

(3) The latches controller will be implemented based on asynchronicity. Since the asynchronous latches controller is full of stateful/oscillating cycles, without any restriction on manufacturing stages or any incompatibility with conventional EDA tools, we show why the adversary is no longer able to engage any form of the SAT attack on this technique.

(4) We thoroughly evaluate and compare the proposed data flow obfuscation with state-of-the-art countermeasures in terms of overhead and security, showing why this new paradigm will be strongly resilient against the existing attacks.

II. BACKGROUND

A. Combinational De-obfuscation

For categories 1, 2, and 3 in Table I, an important threat model assumption is that the attack model is oracle-guided. In an oracle-guided attack model for combinational de-obfuscation, the adversary has access to an unlocked/activated chip (oracle) with open scan chain access, as well as the reverse-engineered yet locked netlist of the oracle. In the SAT attack, for any arbitrary obfuscated combinational logic \(c_{\text{comb,lock}}\), by getting inspiration from the miter circuit used in formal verification, a (distinguishing) miter circuit has been built as \(\text{miter} \equiv c_{\text{comb,lock}}(\text{dip}, k_1) \neq c_{\text{comb,lock}}(\text{dip}, k_2)\), which returns a specific discriminating input pattern (dip) that produces different output for two different keys \(k_1\) and \(k_2\). Then, this dip is queried on the oracle, \(c_{\text{comb, eval}} \leftarrow c_{\text{comb}}(\text{dip})\) and the I/O-constraint...
\( c_{\text{comb\_lock}}(dip, k_1) = c_{\text{comb\_lock}}(dip, k_2) = \text{eval} \) is stored back in the SAT solver and the \textit{miter} circuit would be solved again. When the \textit{miter + constraints} problem has no longer satisfying assignment, it could identify the correct key.

### B. Sequential De-obfuscation

Since the SAT attack is only applicable when the access to the scan chain is open, the studies in category 4 evaluate the security of traditional logic obfuscation techniques [4, 43] while the access to the scan chain is blocked/obfuscated. In this case, the adversary has only access to the PI/PO, and PO would be a function of PI and the state of the circuit, which makes it impossible for the SAT attack to formulate it at once.

To still exploit the combinational SAT attack while the scan access is restricted, few recent studies have engaged unrolling technique as a pre-processing step to formulate the sequential obfuscation using the combinational SAT attack [18, 19, 22, 23]. As shown in Fig. 1, the adversary unrolls the sequential circuits \( \tau \) times. A \( \tau \)-time unrolled circuit is an equivalent combinational model of a sequential circuit for \( \tau \) clock cycles. It takes in \( \tau \) input patterns (as a sequence), and produces \( \tau \) outputs, while the intermediate states are cascaded. After unrolling, similar to the combinational de-obfuscation, the SAT attack would find the sequences of inputs \((i_0, i_1, i_2, \ldots, i_{\tau-1})\), called \textit{distinguishing input sequence} \((\text{dis})\), with two different keys \( k_1 \) and \( k_2 \) such that the outputs \((o_0, o_1, o_2, \ldots, o_{\tau-1})\) will differ. Every time the unrolled \textit{miter} becomes unsatisfiable at some depth \( d \) (no more \( \text{dis} \)), the adversary extends the unrolling until a termination condition.

Termination conditions are \textit{unique completion} \((\text{UC})\): when there is only one key that satisfies the I/O-constraints for unrolled circuit (correct key); \textit{combinational equivalence} \((\text{CE})\): where the transition function is combinatorially equivalent between two duplicated circuits with two keys \( k_1 \) and \( k_2 \) (shadow key), and \textit{unbounded model check} \((\text{UMC})\): if a call to an unbounded model checker \((\tau = \infty)\) concludes that the result is invariant in the reachable state space [18, 19].

The point is that the unrolling step relies on the synchronicity of FFs in the obfuscated sequential circuit. When the sequential circuit is synchronous, moving forward from any arbitrary clock cycle to the next one \((t \rightarrow t+1)\) updates the FFs only once at positive (negative) edges of the clock signal. Hence, in the unrolling step, the combinational parts would be replicated only once per each clock cycle. But, for circuits and systems that \textit{asynchronously} control the data flow in the circuit, the unrolling-based SAT or BMC faces a big obstacle during the unrolling step to build the equivalent combinational model for a specific number of clock cycles.

### C. Asynchronicity

To asynchronously control the data flow in a circuit (partially or fully), one could adopt the asynchronous circuit paradigm. The asynchronous circuits have multiple advantages over synchronous circuits, particularly for newer technology nodes, such as no clock skew problems, robustness towards process variations, as well as advantages in terms of power consumption and electromagnetic emissions [44].

For two main reasons, most designers consider asynchronous circuits as a perilous approach: (1) the lack of electronic design automation (EDA) tools, and (2) opposition to change designers’ mentality towards asynchronicity. However, the ever-increasing attention on these circuits results in introducing powerful synthesis and verification tools for asynchronous circuits [45–47]. It allows any designer to non-disruptively incorporate asynchronicity in an EDA flow, and there is no need for the designer to change the synchronous mentality/structure. As an instance, AnARM is an ultra energy-efficient asynchronous ARM processor that is successfully implemented and fabricated using the STMicroelectronics 28nm technology, using standard cells and conventional CAD tools while achieving a 59% improvement in energy when compared with the ARM Cortex-A7 [48]. As of today, widespread application of asynchronous circuits could be seen in IoTs, NoCs, mixed-signal circuits, etc. [49, 50].

### D. from Synchronicity to Asynchronicity

\textit{Signal transition graph} (STG) is the formal specification of the asynchronous circuits, which is used in most asynchronous synthesis and verification tools [51]. The STG could be drawn from scratch by the designer based on the specification of the design. However, one could use the \textit{desynchronization} paradigm that generates the equivalent asynchronous model of any synchronous circuit. By providing formal proofs of correctness based on the theory of Petri nets [52], the \textit{desynchronization} [53] provides a fully automated flow for building the flow-equivalent asynchronous counterpart.

To build the flow-equivalent asynchronous model of any synchronous circuit using \textit{desynchronization}, as shown in Fig. 2, after removing the clock signal, FFs would be replaced with master \((M)\) and slave \((S)\) latch pairs. All latch enable signals \((en)\) must be controlled using new macros, called asynchronous latches controllers, which uses a handshaking structure \((\text{req}, \text{ack})\) to emulate FFs’ behavior. For example, in four-phase handshaking, as the most prevalent handshaking protocol, \(\phi_1\) is enabling \text{req} by a sender for the a valid data, \(\phi_2\) is enabling \text{ack} by the receiver, acknowledging the arrival of the new data. \(\phi_3\) is lowering (disabling) previous \text{req}, and finally \(\phi_4\) is lowering the corresponded \text{ack}. Handshake signals are not related to a global clock and are based on the local, relative timing relationships between the opening and neighboring latch enable signals. Also, during \textit{desynchronization}, delay elements must be added per each combinational logic \((CL)\) to mimic the delay of all timing
paths and asynchronous latch controller. Also, the first/last latches of the asynchronous part ($m_1$ and $s_3$ in Fig. 2b) that are dealing with other (synchronized) parts of the circuit will be handled by some controlling signals, e.g. a specific state of the circuit, or controlling signals like FFs enables.

It is also worth mentioning that all latches are operating based on their controllers. By using desynchronization, one latch will be enabled when tokens are ready, and will be disabled after receiving the ack corresponded to the new data (lowering req). It will prevent extra propagation when inputs of the latch change, which avoids increasing power consumption. Furthermore, latches controllers are the only added parts when the desynchronization is accomplished; However, we show that since the proposed solution is required to be accomplished on a small part of the circuit, it does not incur large area/resource overhead.

E. Desynchronization

The three main steps of the Desynchronization, which provides a fully automated methodology to build the flow-equivalent asynchronous model of any synchronous circuit are: (1) Converting FFs to $M$ and $S$ latches, with decoupled enable signals (e.g. in Fig. 2b FF₁ is replaced with $M₁$ and $S₁$ whose controllers are $m₁$ and $s₁$). (2) Matched delays generation for combinational logics (CLs), based on their timing path delays. (e.g. $d_{1−4}$ are matched delay for FF₁−₄ to mimic the delay of their timing paths as well as the delay of each asynchronous latch controller ($m_{1−3}$ and $s_{1−3}$)). (3) Implementation of the asynchronous controller of each latch, e.g. ctrls in Fig. 2b, based on the data flow dependencies in the original netlist.

In step 1, after replacing FFs with latches, re-timing is often used as a performance improvement technique [54]. By using re-timing, latches are moved across CLs (e.g. $M₃$ is placed before $CL₄$, $S₁$ is placed after $CL₁$ in Fig. 2).

In step 2, matched delay elements are generated for emulating the timing path delay of their corresponding CLs. These will be connected to corresponding controllers in the next step. In this step, the netlist is synthesized for the target cycle time $T₀$, using a conventional synthesis tool. The $T₀$ is captured using ($T₀ ≥ T_{CQ} + T_{C} + T_{L}$), in which the $T₀$ is a delay between two rising edges of control signal of the latch, $T_{CQ}$ is the delay of local clock propagation through a latch,

$T_{C}$ is the delay of the $CL$, and $T_{L}$ is the latch controller delay. By using this inequality, and based on the delay of critical paths in each $CL$, these matched delays are generated. When $T_{CS}$ are equal in all $CLs$ (balanced timing paths), then the separation time between adjacent rising edges of every local clock equals $T₀$. Also, in any desynchronized circuit, the $i^{th}$ rising transition of a local clock cannot appear later than $(i−1) × T₀$, showing that the temporal behaviors of the desynchronized circuits are also similar to synchronous counterpart [53].

Step 3 implements the asynchronous controller for each latch. These controllers are connected to the controllers of neighboring latches with the delay elements built during step 2. A variety of desynchronization models exist to implement these asynchronous controllers. The behavior of these models can be typically specified using STG, which is a decision-free subclass of Petri nets [52]. An STG, as shown in Fig. 3b, may be defined as a 3-tuple $(Φ, →, I₀)$, where $Φ$ is the set of events, and $→$ are the events enable values (high/low) in asynchronous controllers. $I₀$ corresponds to an arc, which illustrates event transitions, and for a latch controller, it determines changes in latch enable values. $I₀$ is the initial marking, called token, and denotes the initial event signal states. In desynchronization, it is crucial to properly define $I₀$, as the initial tokens, and it is fully dependent to handshaking protocol used for desynchronization [53]. Tokens determine which data is ready. In STGs, as in Petri nets, these tokens could be updated (moved) based on the interaction between different latches. For example, a signal is enabled when all its predecessor arcs are marked with a token. An enabled signal can fire, removing tokens from all its predecessors’ arcs, and populating tokens to its successors’ arcs. For instance, Fig. 3a shows a part of a pipeline with cascaded latches. Fig. 3b depicts an STG representing the behavior of these latches. Over the time, based on the location of data, tokens move around determining which latch will catch a new data.

Without loss of generality, we use semi-decoupled four-phase control for handshaking [55], which represents a good trade-off between simplicity and performance, however, any valid desynchronization latch controller may be used instead [53].

Based on the generalization of semi-decoupled four-phase control, there are four rules imposed on the latch control signals of the STG. Using these four rules, the designer can specify the corresponding STG for any circuit: (1) $a+ → a-$: rising of each signal (each latch enable) should be followed by falling of that signal. (2) $b− → a+$: For latch A (master) to read a new data, latch B (slave) must have completed the read of previous token from A. (3) $a− → b−$: For latch B (slave) to complete the reading of a data token coming from A
must be connected to the circuit depicted in Fig. 4b to build the complete desynchronized counterpart of the synchronous circuit. It is proven in [53] that: (1) A desynchronized circuit never halts (liveness property); and (2) The sequence of data values of a desynchronized circuit is identical to its synchronous counterpart (flow-equivalence).

Also, by using desynchronization, physical design, verification, and design for testability can be accomplished “as is,” using conventional synchronous EDA tools [53]. For instance, for design for testability, a low-frequency clock may be distributed to latches in test mode [56]. For testing the asynchronous controllers, as rising and falling of reqs and acks follow each other, in the presence of a stuck-at fault on either req or ack, either the environment or the circuit will wait forever, and cause a deadlock, which may be easily detected during design for testability. Circuits that have the property that they halt for all faults are called self-testing.

**F. Concept of Data Flow Obfuscation**

Since the sequential SAT attack relies on the synchronous unrolling mechanism, the preserved flow-equivalency after desynchronization, motivates us to propose a new obfuscation paradigm. In general, two circuits could be called flow-equivalent if there is no difference between the sequence of values stored at each latch. The observation is done independently for each latch. As an example of flow-equivalent circuits Fig. 5 demonstrates two flow-equivalent circuits. Fig. 5a shows the synchronous behavior, while Fig. 5b shows the desynchronized behavior. Using this characteristic of asynchronous circuits, in section IV, we show how our proposed obfuscation technique could get benefit from this flow-equivalency concept to introduce a new obfuscation paradigm, called data flow obfuscation.

**III. THREAT MODEL**

Similar to categories 1, 2, and 3 in Table I, we make the following assumption about the adversary capabilities: (1) The adversary can successfully do the reverse-engineering on the chip, and retrieve the gate-level netlist (yet locked). (2) The adversary can purchase an activated/unlocked chip (oracle) from the market. (3) The access to the scan chain of the oracle is not restricted. So, the adversary could apply any query to FFs using SI and read the updated values through SO after one clock cycle (capture mode).

---

*Fig. 4: Desynchronization Flow [53].*  

*Fig. 5: Timing Diagram of Synchronous vs. Asynchronous Circuits [53].*
IV. PROPOSED SCHEME: Data Flow Obfuscation

As discussed previously, in all existing logic obfuscation techniques, the synchronicity is kept intact during manufacturing stages. However, due to the synchronicity, these techniques were vulnerable to unrolling-based SAT or BMC even while the scan chain is restricted. It should be noted that, for the most potent attacks on logic obfuscation, there exists a big inspiration from a formal verification method such that the attack relied on and adopted from the verification method to successfully de-obfuscate circuits, e.g. miter circuit in traditional SAT [8], or BMC/unrolling in sequential SAT [18]. Hence, the main aim of this new obfuscation paradigm is to add ambiguity in a way such that it turns the obfuscated circuit into a completely new form that cannot be modeled using any of the existing formal verification methods. We target part of the data flow in a circuit to be obfuscated using asynchronicity.

When the asynchronicity is used in a circuit, due to the high non-deterministic behavior, it is extremely challenging to come up with an automated approach to establish invariance properties, which are vital in proving the correctness of a circuit with asynchronous parts. There exist a few methods that ease the formal verification in asynchronous parts [57, 58], helping the designers to do formal verification for datapath of asynchronous circuits. However, to prevent any form of easing, in our proposed data flow obfuscation, we target to obfuscate the asynchronous controllers, which is the source of desynchronization with the self-testable feature. Also, since we assume that the scan access must be still fully open, the proposed obfuscation must be in a way that conceals the writing/capturing into/from the storage elements. Hence, in the proposed solution, the controller of latches is obfuscated such that without the correct key, the temporal characteristics of the datapath will be hidden.

A. How Data Flow Obfuscation Works?

The main steps of our proposed data flow obfuscation are:

1. Converting the targeted synchronous part(s) of the circuit to its (their) flow-equivalent asynchronous counterpart using desynchronization described in Sections II-D and II-E.

2. Inserting false paths into the desynchronized circuit. Each false path could be an extra wiring from the output of one latch to any arbitrary combinational logic.

3. Updating the corresponding STG based on the added false paths. For each false path, few extra transitions with initial token must be added to the STG to reflect the changes.

4. Obfuscating the asynchronous latches controller circuit (based on STG w.r.t. the new false paths) by using proposed C*-element that is a key-controlled event-driven AND gate.

Fig. 6 demonstrates step-by-step implementation of the data flow obfuscation on the circuit from Fig. 4a. First, the targeted parts of synchronous circuit (Fig. 4a) are converted to their asynchronous counterpart (Fig. 6a). Then, in the desynchronized circuit a specific number of false paths are inserted (e.g. $D \rightarrow CL_2(E)$ and $G \rightarrow CL_1(A)$ in Fig. 6b). Since the connectivity between latches is altered, the STG should be updated (Fig. 6c). Also, the changes must be reflected into the asynchronous controllers. For instance, before adding the false paths, $F$ was the only predecessor of $A$. So, $R_0$ of $F$ was directly connected to $R_i$ of $A$. However, after adding the false path $G \rightarrow A$, both $F$ and $G$ are the predecessors of $A$. Thus, a C*-element must be added to merge their $req$ signals. The C*-element here implies that latch $A$ may only be opened whenever data from both $G$ and $F$ are ready. However, for any false path like $G \rightarrow A$, it should have no impact on timing when the key is correct. To achieve this, we introduce a C*-element, in which a key-controlled MUX is used to control the C*-element’s inputs. As shown in Fig. 7a, based on the key value, the C*-element input is either $\{A, B\}$ or $\{A, A\}$, and based on the C*-element’s definition, if both inputs are the same, the output will be equal to the identical input pair: $Z = AA + ZA + ZA = A$, meaning that with correct key, the added false paths will have no timing effect.

As shown in Fig. 6, the only obfuscated part in data flow obfuscation is the usage of C*-elements that alters the
behaviour of controllers based on the key value. However, these $C^*$-elements only control the timing (behavior) of latches. Hence, regardless of the key value, each false path is connected directly to one arbitrary chosen $CL$, and could affect its functionality. For instance, for false path $G \rightarrow A$, regardless of the key value ($k_0$), latch $G$ would affect the functionality of $CL_1$, and the key value only controls the time of the act. To avoid this problem, the false paths can be connected to the chosen $CL$ as don’t cares, or non-occurring inputs. Fig. 7b shows these two models for a circuit. As shown, the output of all cases is the same, i.e. $a \lor b$, and the added false path $fp$ does not affect the output in both cases. For instance, in non-occurring$^5$, $fp$ is ANDed with $w_i \land w_j \land w_k$, which is always ZERO, and has no impact on the logic.

B. Shortcomings of False Path Insertion

With inserting only false paths, the data flow obfuscation is, however, vulnerable to re-synthesis and removal attack. Since the false paths are connected to corresponding $CL$s as don’t care or non-occurring, they explicitly have no impact on the circuit’s functionality. Hence, the attacker could re-synthesize the reverse-engineered netlist, and by using logic optimization effort during the synthesis, the false paths will be removed during optimization. Then, the adversary can find some extra elements/connections in the controller that have no corresponding part in the datapath (already removed). So, he/she can distinguish between the original parts and the extra logic added for the false paths in the controller and retrieve the original circuit. So, to combat this issue, we add one more step which adding extra false latches on false paths.

C. Adding False Latches on the False Paths

We updated and added one more step in our proposed data flow obfuscation to support adding false latches:

1. (Same Step) desynchronization + re-timing (relocate).
2. (Same Step) Inserting false paths into the circuit.
3. (New Step) Inserting false latches (MIS pairs) on the false paths + an asynchronous controller for each added false latch to control its behavior.
4. (Same Step) Updating the STG based on new insertions.
5. (Same Step) Obfuscating the controller via $C^*$ elements.

Inserting pairs of false latches in false paths allows us to control the logic value of these paths. So, there is no longer a need to add false paths as don’t care or non-occurring, and the re-synthesis and removal attack is no longer a valid attack. The concept of insertion of false paths is visualized in Fig. 8. Similar to the previous example, first, the circuit must be converted to its asynchronous counterpart (desynchronized). Fig. 8a shows the asynchronous model of our simple circuit from Fig. 4a. After that, one false path is added from $CL_2(B)$ to $CL_1(A)$, then a pair of master and slave latches ($I$ and $H$), are added in this path (Fig. 8b). Based on these changes, the STG is updated (Fig. 8c). Compared to the STG of the previous example, not only new transitions are added, the STG has new nodes describing the events of new false latches. Finally, these updates must be reflected into the obfuscated asynchronous circuit (Fig. 8d). Note that extra controller modules are added for false latches. Also, the key-controlled gates must be added to properly control the behavior of latches $H$ and $I$. When the key is correct, the added false path must have a value that does not affect the functionality of $CL_1$. For this purpose, the behavior of the false latches is controlled using $k_{0-2}$. So, while the $k_{0-2}$ is correct (000 in this case), the AND gates mask the handshaking of $H$ and $I$ with their neighboring latches. Hence, these latches are disabled (and no new data will be captured in them). So, the initial value of these latches will be kept intact, and will be used as the new input of $CL_1$. In $CL_1$, based on the initial value of these latches, this false path will be connected to an arbitrary gate. (e.g., with initial value 0, it could be connected to an OR or XOR gate). However, while the key is not correct, the behavior of these latches would be changed repeatedly, resulting in corrupting the functionality of $CL_1$.

Additionally, false {paths + latches} must not affect their neighboring latches when the key value is correct (e.g. latches $H$ and $I$ must have no effect on $A$ and $B$ in Fig. 8d). This is achieved by adding two $C^*$-element before $A$ and $B$, controlled by $k_3$ and $k_4$ respectively to effectively eliminate this temporal relation (e.g. in path $I \rightarrow A$, $C^*$-element skips the effect of the behavior of $I$ on the behavior of $A$).

D. Key Classification

The keys added to asynchronous latches controllers, will be categorized into two main groups: (1) handshake-in keys: keys that control the impact of incoming signals to false latches ($k_{0-2}$ in Fig. 8d), (2) handshake-out keys: keys that control the impact of outgoing signals from false latches ($k_{3-4}$ in Fig.
Based on the value of these two groups, different scenarios could happen: (1) **Correct Functionality**: While both groups are correct. In this case, similar to the timing diagram depicted in Fig. 9a, the firing of latches alternates appropriately. (2) **Halt in data flow**: While handshake-in keys are correct, but handshake-out keys are incorrect, halts will happen (e.g., if \( k_0 \_2 = 000 \) and \( k_3 \_4 \neq 00 \), \( H \) and \( I \) would halt). As shown in Fig. 9b, after latch \( H \) controller (\( h \)) is halted, more halts are happened in other paths and results in a complete deadlock in the whole circuit. (3) **Incorrect Functionality**: While the handshake-in keys are incorrect, regardless of the handshake-out keys, the function will be incorrect.

### V. Security/Testability Analysis

#### A. Security vs. the SAT Attack

Since the adversary has access to the scan chain in data flow obfuscation, he/she is able to apply combinational des-obfuscation for any accessible part of the circuit using the SAT attack. However, for two important reasons, the traditional SAT attack cannot be applied on data flow obfuscated circuit: (1) In the SAT attack, it is crucial to know the exact time of writing/capturing into/from the scan chain; But, in data flow obfuscation, this timing is controlled (locked) by an asynchronous controller. The adversary cannot determine when he/she must write into the scan, and when the updated data is ready to be observed. (2) Due to the nature of asynchronous controllers, the latch enable controller consists of many stateful cycles. The SAT solver works perfectly fine if the circuit is a directed acyclic graph (DAG), and only structural cycles could be analyzed using a pre-processing engine before running the SAT solver. Depending on whether the cycle is oscillating or stateful, the SAT solver will either be trapped in an infinite loop or will return UNSAT. Moreover, in attacks such as BeSAT [40] that can track and detect non-structural cycles, the very first assumption is that the circuit has no stateful combinational cycle by itself.

#### B. Vs. Sequential SAT Attack

The adversary may attempt to engage either an unrolling-SAT attack or SAT integrated with BMC (SAT-BMC) by creating the unfolded combinational equivalent circuit to find \( dises \). The length of \( dises \) determines the number of unrolling required before running the SAT solver. When the circuit is synchronous, per each clock cycle, the FFs will be updated only once. So, the adversary can replicate whole \( CLs \) iteratively (per each clock cycle) to build the unrolled circuit. However, when we use asynchronicity in data flow obfuscation, the adversary needs to know the list of enabled latches continuously and cycle-accurately to unroll those parts that are triggered with new latched data. But, in the data flow obfuscation paradigm, the asynchronous controller that determines which latches must be enabled/disabled is obfuscated. So, the adversary cannot build the unrolled circuit to still get the benefit of the SAT attack, and thus, the sequential SAT attack cannot be applied to this technique.

### C. Vs. Re-Synch. + Sequential SAT Attack

Since the sequential SAT attack cannot be applied directly to data flow obfuscation, the adversary might add a pre-processing step, such as re-synchronization, to make this attack valid. In re-synchronization, which is the reverse of desynchronization, the asynchronous controller is removed, \( M \) and \( S \) latches are merged as FFs, and all FFs are connected to a global clock. However, for a few reasons, re-synchronization of obfuscated desynchronized netlist is not possible:

First, since each controller requires a local clock tree in asynchronous circuits, and these local trees do not have the same delay [59], the adversary needs to confirm two constraints below, which contradict each other: (1) Theoretically, the adversary must use a clock period larger than any delay element, to avoid metastability happening in the asynchronous controllers from the delay element. The delay constraints of an asynchronous circuit could be modeled using the following formulas obtained from Fig. 10:

\[
D_{R_t} + T_{r_{t+1}} + D_{L_t+1} > D_{L_t} + T_{c_t} + D_{P_t} + T_{s_{t+1}}
\]  

(1)

\[
D_{A_t} + T_{a_t} + D_{L_t} + T_{c_t} + D_{P_t} > D_{L_{t+1}} + T_{h_{t+1}}
\]

(2)

(2) But, the adversary must use a very small clock period to utilize the delay element as a time offset. Hence, based on these two constraints, the timing correctness conditions cannot be satisfied. Also, the timing constraints between the datapath and the asynchronous controller must be preserved making it more challenging [59].

Second, as a step during re-synchronization, the attacker must analyze every connectivity in the netlist, and effectively create a mapping problem using bipartite graphs between pairs of \( \{ M, S \} \) to FFs. To accomplish this, the attacker must have prior knowledge of design methodology used for creating the asynchronous circuit (2-phase latches or 3-phase latches, handshaking protocol, initial marking (tokens), etc.). Even while the adversary has access to this prior knowledge, since the connectivity is obfuscated using false paths, the false mapping will be added to this bipartite graph, leading to failure of correct matching between \( Ms \) and \( Ss \).
Additionally, as shown in Fig. 2 and Fig. 4, during desynchronization flow, re-timing has been engaged in data flow obfuscation by moving parts of CLs before/after the latches. By doing so, re-synchronization cannot be accomplished directly. Re-timed asynchronous circuit can be converted to a 2-phase non-overlapping synchronous design, with two clock signals, \( \phi_1 \) and \( \phi_2 \). However, false paths with extra latches makes this 2-phase non-overlapping synchronous netlist malfunction. For example, latches \( H \) and \( I \) in Fig. 8b would be connected to clock signal \( \phi_1 \) and \( \phi_2 \) (non-overlapping clock signals). By connecting these two false latches to clock signals, their values would be updated which alters the functionality of \( CL_1 \).

**D. Vs. Structural-based Attacks**

The attacker might try to guess the value of the keys based on the overall structure of the locked netlist. For instance, all handshaking signals to/from latches \( H \) and \( I \) in Fig. 8d are controlled using \( C^* \)-element and key-gates (ANDs). Hence, the attacker might guess that this pair of latches are false latches located on a false path. So, the value of the keys can be retrieved easily. However, to avoid such circumstances, these key-gates and \( C^* \)-element will be added for a set of arbitrary (actual) latches in the netlist. For example, in Fig. 8d, the same key-gates are added between \( C \) and \( D \). However, they always must be active. Also, the original \( C \)-element before \( C \) and \( E \) could be replaced with \( C^* \)-element. So, unlike \( C^* \)-element before \( A \) and \( B \), in which only one of the inputs is valid, in these \( C^* \)-elements, both inputs are valid. By using this simple mechanism, the attacker cannot start guessing/detecting the false \{paths + latches\} based on the location/type of key gates. Fig. 11 shows a more complicated instance, in which all \( C \)-element are replaced with \( C^* \)-element. Also, similar to the latches \( H \) and \( I \), \( C^* \)-element and key-gates (ANDs) are used for \{\( A, B \)\} and \{\( C, D \)\}. So, the attacker cannot apply any form of key-guessing structure to find the false \{paths + latches\}.

**E. Vs. Other State-of-the-Art Attacks**

As was shown in Table I, there exist many attacks on different logic obfuscation techniques, each is modeled to break one or more specific techniques. However, Table II explains why none of these attacks is applicable to the proposed solution. The biggest advantage of the proposed data flow obfuscation is the usage of extensive non-determinism of asynchronicity for obfuscation purposes. In data flow obfuscation, the main source of this non-determinism, which is the asynchronous controller, is the main target of obfuscation. Obfuscating an asynchronous controller makes every step of simplification dependent on the key value, and it extremely boosts up the state space in the asynchronous part. This implies the difficulty of attacking the proposed data flow obfuscation, where it requires an extensive (likely impossible) investigation on how the existing formal verification methods might be fitted and useful to be adopted in this case.

**F. Testability of Data Flow Obfuscation**

As discussed previously, the handshaking asynchronous controller is a self-testing circuit. However, since we use the halt in false paths for logic locking purposes, data flow obfuscation prevents the self-testing and contradict this...
TABLE III: Specifications of the Benchmark Circuits (ISCAS’89, ITC’99, and well-known ASICs/microprocessors).

| Small Circuit: s258 s526 s1423 s5378 s9234 s13207 s15850 s35932 s38584 |
|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
| # of Inputs      | 3                | 3                | 17               | 35               | 36               | 62               | 77               | 35               | 38               |
| # of Outputs     | 6                | 6                | 5                | 49               | 39               | 152              | 150              | 320              | 304              |
| # of Gates       | 119              | 193              | 657              | 2779             | 5597             | 7951             | 9772             | 16165            | 18253            |
| # of FFs         | 14               | 21               | 74               | 179              | 211              | 638              | 534              | 1728             | 1426             |
| Large Circuit:   | b17              | b18              | b19              | MC8051 AES-GCM SPARC |
| # of Inputs      | 37               | 37               | 24               | 52               | 116              | 95               |
| # of Outputs     | 97               | 23               | 30               | 112              | 15               | 108              |
| # of Gates       | ~28K             | ~95K             | ~190K            | ~6.6K            | ~49.5K           | 233K             |
| # of FFs         | ~1.5K            | ~3.3K            | ~6.6K            | ~1K              | ~5.1K            | 12K              |

property. To protect against an untrusted test, this contradiction enforces the designer to use an incorrect key to keep the self-testing property of these circuits. For instance, \( k_{0-2} \) in Fig. 8d must be 111 to avoid any halt (the correct key is 000).

As another example, similarly, \( k_5 \) in Fig. 8d must be 1 to avoid halt on the other path (But in this case, the correct key is 1). Also, since \( C^* \)-element does not create a halt in any path, keys connected to \( C^* \)-element could be an arbitrary value to choose a path in latches controllers. It shows that there is no relation, e.g., bit flipping, between the correct key and key used for the test. Using incorrect key allows false latches located in the false paths to be updated. Hence, false paths can affect the CLs’ functionality. Although the designer can generate test patterns that avoid making them driving, since the incorrect key only adds false paths to the original netlist, few more test patterns are required to test these paths, which has no impact on test patterns generated for original parts of the netlist. Hence, there is no restriction for test pattern generation.

VI. EXPERIMENTAL RESULTS

We evaluate the data flow obfuscation over three sets of benchmark circuits, all listed in Table III. The experiments are all executed on a 24-core Intel Xeon processors running at 2.4GHz with 256 GB of RAM. Area, power, and delay overhead of the data flow obfuscation are obtained using conventional Synopsys Design Compiler along with Synopsys generic 32nm library. We evaluate the security/overhead of the data flow obfuscation based on: (1) obfuscation overhead: Selection, desynchronization, and insertion of false {paths + latches} depend on the circuit size. (2) key size: Regardless of the circuit size, for a key size, a nearly fixed part of a circuit will be selected, desynchronized, and false {paths + latches} will be inserted.

A. Modeling of Attacks on Data Flow Obfuscation

Since the proposed data flow is dependent to the locked asynchronous controller, the timing of writing/capturing into latches is hidden, and the traditional SAT attack does not work even while the scan chain is available. So, to evaluate the security of the data flow obfuscation, we deploy two new versions of sequential SAT: (1) S_SAT: Resync + BMC + SAT, (2) S_BeSAT/S_icySAT: BMC + BeSAT/icySAT [14]/[40].

Regarding the former version of the deployed attack, due to the failure of unrolling on desynchronized circuits, we need a pre-processing step to re-produce the re-synchronized version of the obfuscated circuit. We discussed in Section V-C that the exact re-synchronization is almost impossible. In this section, to validate our claim, regardless of the timing criteria, we developed an intuitive re-synchronization technique. The steps of the re-synchronization are as follows: (1) Removing the obfuscated asynchronous controller; (2) Merging each pair of \( M \) and \( S \) latches based on the connectivity of them (moving them across CLs). (3) Replacing each pair of \( M \) and \( S \) latches with a FF. (4) Connecting FFs to a synchronous clock signal. (5) Adding an extra key-controlled MUX for each path that comes from the output of FFs. (For each MUX, an input comes from the output of the FF, and one input is an extra key. The selector of the MUX is another key input.) Using these 5 steps, Fig. 12 shows the re-synchronized version of the obfuscated circuit from Fig. 8b. Now, this re-synchronized version could be the input of the Sequential SAT attack. By using this model, if a path is a false path, then the logic value of this path is always fixed. So, the MUX must select the extra key input with corresponded value; However, if a path is an actual one, the other input of the MUX must be selected.

Regarding the latter versions of the deployed attack, on the other hand, no re-synchronization has been used, and since the asynchronous controller is in place with lots of combinational cycles, we replaced the traditional SAT attack with existing cyclic-SAT attacks, i.e. BeSAT and icySAT [14, 40].

B. Attack Results

Table IV shows the results of two attacks. For all cases, both attacks failed to break the obfuscated desynchronized circuit. The result of the attacks, in both versions, might return a wrong key, might return UNSAT, or might trap in an infinite loop. These three scenarios happen for a few main reasons: (1) regarding the wrong key (w/k) and UNSAT in S_SAT, the unrolling could not build the correct unrolled version due to asynchronicity; (2) regarding facing an infinite loop in S_BeSAT and S_icySAT, all are because of facing lots of stateful cycles in asynchronous controller; and (3) regarding the UNSAT in S_BeSAT and S_icySAT, before facing an infinite loop, the solver is trapped in a wrong decision leading to UNSAT (because of incorrect formulation).

As shown in Table IV, in some rare cases, we see some numbers are struck out and replaced with w/k and UNSAT when we apply the S_SAT on re-synchronized circuits. In these cases, the S_SAT was able to find the correct key values. However, we found that re-timing did not relocate the latches after desynchronization for such cases. So, we force the re-timing step to do a minor relocation for set of latches to eliminate the possibility of applying any form of
re-synchronization. In this case, after enforcing those minor relocation, the S_SAT fails to break them. Also, for some cases, we face time-out (10^5 seconds), implicitly showing the complexity of SAT circuit. For all other cases, since we remove all stateful combinational cycles during re-synch., we only faced with w/k or UNSAT. Table IV implies that the existing attacks cannot formulate the proposed solution properly to break it, regardless of the size/portion of the circuit, and regardless of the number of false paths inserted into the circuit.

Based on the two obfuscation metrics, we evaluated these deployed attacks in 5 different scenarios, in which a specific value for one of the metrics has been fixed. Table V shows that for each scenario with a fixed metric, what the value of the other metric is, which helps us to have an estimated relationship between these two metrics.

C. Area/Power/Delay Overhead Comparison

In this section, we evaluate the post-synthesis overhead of our data flow obfuscation. Table VI compares the power, performance (delay), and the area (PPA) of the original vs. obfuscated circuits while the obfuscation overhead is set to 10%. 10% obfuscation overhead means that 10% of all FFs in a circuit must be converted to latches using desynchronization.

Also, for any obfuscation overhead percentage, the number of extra \{paths + latches\} is set to be less than 10% of the total latches. Further, the actual latches that are obfuscated using the same key gates (to prevent any key-guessing or structural attacks) are set to be less than 10% of the total latches. For example, for b17 with \~1.5K FFs, for 10% obfuscation overhead, we replace 150 FFs with latches; we insert up to 15 false \{paths + latches\}, and up to 15 actual latches are obfuscated using the same key gates.

The overhead of desynchronization is dominant while the ratio of FFs to the total number of gates is higher in the original netlist. For instance, for s13207, whose FFs' ratio to all gates is 638/7951 = 8.03\%, the area overhead is 6.72\%. However, in s9234 with a ratio of 3.76\%, the area overhead is only 4.28\%. Table VII demonstrates the area breakdown of some of the circuits when the obfuscation overhead is 10%.

Regarding the delay overhead, since re-timing is used during desynchronization, in some cases we even achieved slight delay improvement. However, in some cases, it imposes only a very slight difference in cycle time by up to 8%. Regarding the power overhead, due to moving from edge-triggered design to level-triggered, the power overhead is less compared to area overhead. As seen in Table VI, our data flow obfuscation
paradigm increased the power consumption by up to 10%.

Table VIII compares the PPA of the original circuits vs. obfuscated circuits when the key size is 200. As shown in Fig. 11, and as implied in Table V, in data flow obfuscation, for each extra [paths + latches], as well as for any actual latch that are obfuscated to disable key-guessing, 3-5 keys could be added. So, when the key size is 200, regardless of the size of the circuit, 50-60 latches are needed. Accordingly, as the size of the key is 200, the area overhead is much higher in small circuits. However, for larger circuits, the ratio of false latches compared to the size of the circuit is significantly low, and since the real applications (ICs) are far larger than small circuits listed in Table III, the area overhead is low in this approach. For instance, in AES-GCM, the area overhead is even less than 1% (0.4%). To reflect a better evaluation of overhead, in Table IX, we compare the overhead of data flow obfuscation when the key size is set to 200 with state-of-the-art logic obfuscation techniques. As shown, on average, the overhead incurred by data flow obfuscation is much lower compared to almost all techniques. In some techniques, the overhead of one metric might be better than that of data flow obfuscation, but on average, it could be concluded that the overhead of the proposed technique is completely acceptable.

Since the strength of the data flow obfuscation does not depend on the number of [paths + latches], we could add as much as the key size (e.g. ≥ 64) to prevent breaking them using attacks like brute-force.

D. Comparison with State-of-the-art

Most recently, a new study has evaluated the possibility of latch-based architecture as a new means of logic obfuscation [60]. In this study, as shown in Fig. 13, key-programmable latches could be used as: (1) regular storage elements (FFs that are replaced with green latches subjected to re-timing), (2) programmable logic decays (red latches/CLS) with constant output (always zero with no driving effect), and (3) programmable path delay decay for delay manipulation (yellow latches). However, unlike our proposed data flow obfuscation, it is still fully dependent on the clock signal (reset and enable of latches are a function of the clock signal) allowing us to convert this solution to a synchronous obfuscated problem. Clock-dependent latches operate as storage elements with synchronized gated clock, and due to this synchronicity, by integration with two pre-processing steps, it still could be modeled and broken using sequential SAT integrated with cyclic model: (step 1) Generating a single-clock synchronized locked circuit using a generalized/automated model, and (step 2) Detection of (some) programmable logic decays with constant output through (pseudo-exhaustive) test patterns on testable points of oracle.

Unlike latch-based logic locking that uses BMC with two copies of each circuit per each clock cycle (due to level-triggering of latches) [60], we first generate a single-clock model to avoid this duplication per cycle. Fig. 14 shows we could build a single-clock fully synchronized circuit from the latch-based logic locked circuit. Based on the modes of latches demonstrated in Fig.13c, Latches are replaced with a FF with two MUXes (one 2-to-1 preceding and one 4-to-1 following). The 4-to-1 MUX builds all three modes illustrated in Fig.13c, and the 2-to-1 MUX is for keeping FF values when clk is not triggering (latching). Also, a 2-to-1 MUX will be added before each neighboring FF (immediate neighbors of latches). Now, all FFs are connected to a low frequency generalized clock signal (clkg). The selector of 2-to-1 MUXes of FFs corresponded to latches will be connected to the original clock signal (reset and enable of latches are a function of the clock signal). This generalized model could be used for different scenarios with more complexities, such as multi-clock systems, and circuits with gated clock, all could be synchronized using generalized clock signal clkg [61]. It allows us to use BMC with ONLY one copy of circuit per each clock cycle, which improves the scalability significantly.

Also, since programmable logic decays (red latches and red CLS) always generate constant output (zero output), and since...

---

**TABLE VIII:** Data Flow Obfuscation Overhead (Key Size = 200).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Area ( \Delta m^2 )</th>
<th>Max Delay ( n_s )</th>
<th>Power ( \Delta v' )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original locked</td>
<td>original locked</td>
<td>original locked</td>
</tr>
<tr>
<td>s298</td>
<td>152.1 618.6 306.71%</td>
<td>0.31 0.32 3.7%</td>
<td>14.17 26.41 88.4%</td>
</tr>
<tr>
<td>s526</td>
<td>284.3 722.8 154.24%</td>
<td>0.26 0.28 8.2%</td>
<td>13.34 22.76 70.6%</td>
</tr>
<tr>
<td>s1423</td>
<td>1018.7 1462.2 43.54%</td>
<td>1.22 1.12 -8.3%</td>
<td>44.84 51.05 13.9%</td>
</tr>
<tr>
<td>s5378</td>
<td>2181.5 2640.1 21.02%</td>
<td>0.62 0.64 2.9%</td>
<td>103.2 114.48 10.9%</td>
</tr>
<tr>
<td>s9234</td>
<td>2895.1 3306.6 14.21%</td>
<td>0.38 0.36 -6.3%</td>
<td>131.3 141.35 7.7%</td>
</tr>
<tr>
<td>s13207</td>
<td>5023.5 5493 9.35%</td>
<td>1.28 1.25 -2.0%</td>
<td>214.9 227.28 5.9%</td>
</tr>
<tr>
<td>s15850</td>
<td>6077.1 6521.6 7.31%</td>
<td>1.25 1.16 -6.8%</td>
<td>272.6 281.37 3.2%</td>
</tr>
<tr>
<td>s35932</td>
<td>15413.4 15844.9 2.80%</td>
<td>1.15 1.25 8.6%</td>
<td>1699.4 1648.8 2.4%</td>
</tr>
<tr>
<td>s38584</td>
<td>23186.2 23583.1 2.01%</td>
<td>1.14 1.18 3.7%</td>
<td>1786.9 1814.7 1.6%</td>
</tr>
</tbody>
</table>

**TABLE IX:** Overhead Comparison between Data Flow Obfuscation vs. State-of-the-art Obfuscation Techniques.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>CAT 1</th>
<th>CAT 2</th>
<th>CAT 3</th>
<th>CAT 4</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLL</td>
<td>26%</td>
<td>5.3%</td>
<td>24.4%</td>
<td>5%</td>
<td>2.8%</td>
</tr>
<tr>
<td>DPLL</td>
<td>39.6%</td>
<td>37.9%</td>
<td>37.9%</td>
<td>14.7%</td>
<td>2.4%</td>
</tr>
<tr>
<td>CYCLE</td>
<td>39.6%</td>
<td>37.9%</td>
<td>37.9%</td>
<td>14.7%</td>
<td>2.4%</td>
</tr>
<tr>
<td>Obf</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Full</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Lock</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Lock</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Inter</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>R-DFS</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>+SLL</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>+TRLL</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Delay</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Power</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Area</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Delay</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Power</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Area</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Delay</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
<tr>
<td>Power</td>
<td>29.8%</td>
<td>47.6%</td>
<td>37.7%</td>
<td>13.5%</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

**Fig. 13:** Latch-based Logic Locking Scheme [60].

(a) (b) (c)
TABLE X: Runtime of Re-synchronization + Decoy COI reduction + Sequential SAT Integrated with BeSAT (S\textunderscore BeSAT) on the existing latch-based logic locking with Key Size = 20, 50, 100, 200.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>key size = 20</th>
<th>key size = 50</th>
<th>key size = 100</th>
<th>key size = 200</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1423</td>
<td>468.5</td>
<td>521.6</td>
<td>321.1</td>
<td>1450.8</td>
</tr>
<tr>
<td>s5378</td>
<td>1422.1</td>
<td>1475.9</td>
<td>1855.6</td>
<td>2991.3</td>
</tr>
<tr>
<td>s15850</td>
<td>225.8</td>
<td>3006.5</td>
<td>2284.5</td>
<td>4712.6</td>
</tr>
<tr>
<td>s35932</td>
<td>3824.5</td>
<td>2071.9</td>
<td>10253.4</td>
<td>11452.8</td>
</tr>
</tbody>
</table>

timeout: 10^5 Seconds ≈ one day (Stop the attack when time reaches timeout)

testable pin are dedicated for latches (using extra MUXes and duplicate FFs) in the existing latch-based approach [60], the adversary would be able to apply test patterns (stuck-at-fault or pseudo-exhaustive) on testable points at Cone-Of-Influence (COI) of oracle to detect latches with constant (zero) output. For some fundamental reasons, the programmable logic decoys could not be large enough to make this test infeasible: (1) programmable logic decoys are hardware overhead, which must be limited, (2) these logic decoys add difficulties to P&R which compromises the performance, and (3) it should not have an impact on maximum frequency of the circuit before adding obfuscation. Detecting these latches helps reducing the cone-of-influence, and consequently the SAT circuit before running the sequential SAT attack when it is integrated with these two pre-processing steps. However, since our proposed data flow obfuscation is truly desynchronized (token-based with no dependency to a clock signal), this form of re-synchronization is not applicable to it.

Table XI also shows some major advantages of the proposed solution against existing latch-based logic locking [60]. For example, we use a self-testable asynchronous controller; however, latch-based adds extra MUXes/FFs to make the latches testable, resulting in extra overhead. We add false latches pair by pair, which makes any structural/functional analysis exponentially harder (any pair of neighboring latches is a point of analysis); However, latch-based adds decoy latches one by one, which allows analyzing them linearly (each latch is a point of analysis). In the proposed scheme, the latch enables controller consists of many stateful cycles that boost the difficulties for the adversary (no cyclic modeling). However, only easy-to-track structural cycles might be added into the existing latch-based logic locking when the key is not correct. Also, the overhead is much higher in the existing approach due to adding programmable logic decoys.

**VII. CONCLUSION**

To combat state-of-the-art attacks on logic obfuscation, in particular the SAT attack and the sequential SAT, in this paper, we introduced a new obfuscation paradigm called data flow obfuscation. By exploiting the concept of asynchronicity, in data flow obfuscation, we show how the flow of the data could be obfuscated in any arbitrary circuit. In data flow obfuscation, we engage false \{paths + latches\} using the asynchronous structure to control the flow of data in specific timing paths. Using this mechanism, we show that the SAT attack has no longer an advantage for the adversary even while the scan access is not restricted. Also, we showed that how asynchronicity combat the sequential SAT attack by invalidating the unrolling step in these attacks. We comprehensively investigated the effectiveness of this new obfuscation paradigm over wide-range benchmark families. Our experiments showed the resiliency of this new paradigm against all existing attacks at significantly low overhead.

**REFERENCES**

Kimia Zamiri Azar received her B.Sc. degree in computer engineering from Khajeh Nasir University in Tehran, Iran. She received her M.Sc. degree in computer engineering from the Shahid Beheshti University, Tehran, Iran in 2015. She is currently a Ph.D. student at the Electrical and Computer Engineering department at George Mason University. Her research interests are in the areas of hardware security and trust, security for supply chain, and VLSI design and test.

Hadi Mardani Kamali received his B.Sc. degree in computer engineering from the University of Tehran, Iran in 2015. He has worked earlier on telecommunication devices for SRD in Iran. He is currently a Ph.D. candidate at the Electrical and Computer Engineering department at George Mason University. His research interests are in the areas of approximate computing, low power design, hardware-level functional safety, and security.

Houman Homayoun received the Ph.D. degree from the Department of Computer Science, University of California at Irvine, Irvine, CA, USA, in 2018. He is currently an Associate Professor at the Electrical and Computer Engineering Department, University of California Davis, CA, USA. Since 2017 he has been serving as an Associate Editor of IEEE Transactions on VLSI. He was the technical program co-chair of GLSVLSI 2018 and the general chair of the 2019 GLSVLSI conference.

Christos P Sotiriou received the Ph.D. degree in computer science from the University of Edinburgh, Edinburgh, Scotland, in 2001. He is currently an Associate Professor at the ECE Department, University of Thessaly, Greece. His research interests include CAD algorithms and tools, and design methods and approaches for asynchronous timing. He served as General Chair of the International Symposium on Advanced Research in Asynchronous Circuits and Systems in many years.

Avesta Sasan received his B.Sc. in Computer Engineering from the University of California Irvine in 2005. He then received his M.Sc. and his Ph.D. in Electrical and Computer Engineering from the University of California Irvine in 2006 and 2010 respectively. Dr. Sasan then worked at the industry in Broadcom and Qualcomm Co. till 2016. He joined George Mason University in 2016, where he is currently serving as an Associate Professor in the Department of Electrical and Computer Engineering.