On the Complexity of Generating Gate Level Information Flow Tracking Logic

Wei Hu, Jason Oberg, *Student Member, IEEE*, Ali Irturk, *Member, IEEE*, Mohit Tiwari, Timothy Sherwood, *Member, IEEE*, Dejun Mu, and Ryan Kastner, *Member, IEEE*

***Abstract*—Hardware-based side channels are known to expose hard-to-detect security holes enabling attackers to get a foothold into the system to perform malicious activities. Despite this fact, security is rarely accounted for in hardware design flows. As a result, security holes are often only identified after significant damage has been inflicted. Recently, gate level information flow tracking (GLIFT) has been proposed to verify information flow security at the level of Boolean gates. GLIFT is able to detect all logical flows including hardware specific timing channels, which is useful for ensuring properties related to confidentiality and integrity and can even provide real-time guarantees on system behavior. GLIFT can be integrated into the standard hardware design, testing and verification process to eliminate unintended information flows in the target design. However, generating GLIFT logic is a difficult problem due to its inherent complexity and the potential losses in precision. This paper provides a formal basis for deriving GLIFT logic which includes a proof on the NP-completeness of generating precise GLIFT logic and a formal analysis of the complexity and precision of various GLIFT logic generation algorithms. Experimental results using *IWLS* benchmarks provide a practical understanding of the computational complexity.**

***Index Terms*—Algorithm design and analysis, Boolean functions, computational complexity, gate level information flow tracking, information security.**

I. Introduction

HIGH-ASSURANCE systems such as those found in critical infrastructures and medical devices have strict requirements on information security. For example, the Boeing 787 uses a shared data network between the passenger network and systems critical to the safe operation of the plane [1]. Unintended data movement from the user to the flight control network could potentially violate the integrity of the flight control system, and there should be methods that ensure this never occurs. Medical devices are another example of systems that require high assurance. There are published attacks on insulin pumps [2] and pacemakers [3] which describe how to compromise patient privacy and even safety. Unfortunately, developing high assurance systems is a costly endeavor. As an example, developing a high assurance real-time operating system required extensive third party analysis [4], costing millions of dollars [5], and taking years to complete [6].

Two common policies used to uphold information security are *discretionary access control* and *information flow control* (IFC). Access control mechanisms are often effective, but lack the transitivity to monitor the movement of information. In other words, these techniques provide a policy for how some object $\mathbf{B}$ can access some object $\mathbf{A}$, but they specify nothing about how $\mathbf{B}$ will use $\mathbf{A}$'s data once it receives it. IFC, on the other hand, is transitive in that it specifies a strict policy about where an object's data can flow. In terms of the shared data network on the Boeing 787, an access control mechanism would specify how the user and flight control systems can access some common resource, but states nothing about how the resource uses any received information from either party. An information flow control policy, however, could guarantee that no information flows from the user to flight control network, even through shared resource. This strict policy provides strong assurance for confidentiality [7] (e.g., that secret information will not leak to the unintended places) and for integrity [8] (e.g., that high-assurance components will not be affected by untrusted data). To uphold an IFC policy that encapsulates data integrity or confidentiality, a method known as information flow tracking (IFT) is commonly used to provide strict bounds on the movement of information.

Information flow tracking is a method to enforce IFC that associates a label with data and tracks the movement of this label through the system. IFT has been deployed at many levels of the system stack including in the programming language [9], operating system [10], [11], instruction set and micro-architecture [12], [13] and runtime system [14], [15]. Unfortunately, these methods ignore hardware specific timing channels. Such hardware specific timing channels are known to leak secret encryption keys in stateful elements such as caches [16] and branch predictors [17]. Further, timing flows from untrusted entities to ones of high integrity (e.g., from the user to flight control network) can cause violations in real-time constraints, greatly hindering the intended operation of a system or even rendering the critical system useless.

Manuscript received December 12, 2011; accepted February 09, 2012. Date of publication February 28, 2012; date of current version May 08, 2012. This work was supported by NSF Grant 0910581. The work of J. Oberg was supported by an NSF Graduate Research Fellowship. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Ramesh Karri.

W. Hu and D. Mu are with the School of Automation, Northwestern Polytechnical University, Xi'an 710072, Shaanxi, China (e-mail: vinnie@mail.nwpu.edu.cn; mudejun@nwpu.edu.cn).

J. Oberg, A. Irturk, and R. Kastner are with the Department of Computer Science and Engineering, University of California, San Diego, CA 92093 USA (e-mail: jkoberg@cs.ucsd.edu; airturk@cs.ucsd.edu; kastner@cs.ucsd.edu).

M. Tiwari is with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720 USA (e-mail: tiwari@eecs.berkeley.edu).

T. Sherwood is with the Department of Computer Science, University of California, Santa Barbara, CA 93106 USA (e-mail: sherwood@cs.ucsb.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2012.2189105

To prevent hardware specific timing channels, the most commonly used techniques are physical isolation and “clock fuzzing” [18]. Physical isolation, as the name suggests, requires that mix-trusted subsystems be completely isolated from each other. Although surely information flow secure, this prevents the two systems from sharing any common resource and ultimately leads to hardware overheads due to replication. “Clock fuzzing” is an *ad hoc* attempt to mitigate the likelihood of extracting information from a timing channel by introducing entropy into the channel. This method, however, only reduces the bandwidth of the channel (by creating a lower signal to noise ratio) and provides no formal guarantees about eliminating a timing channel completely [19].

Gate level information flow tracking (GLIFT) [20] is a technique that can identify hardware specific timing channels in addition to other explicit and implicit information flows allowing formal guarantees on their (non)existence. GLIFT associates a single bit label with every data bit in the system and monitors the movement of each individual bit of state in hardware as they flow through Boolean gates. At this low level of abstraction, all logical information flows are detectable, including those through timing channels. It is the only formal way of detecting hardware specific timing channels to the best of our knowledge. It can also detect other implicit and explicit information flows that are monitored by IFT methods at higher levels of the system stack.

Previous work has shown that GLIFT can be used to guarantee strict isolation between different execution contexts of software running on a processor while communicating with various I/O devices. An execution lease architecture was developed to dole out microarchitectural state to untrusted execution contexts [21]. To prevent information flows, even those through hardware specific timing channels, the architecture constrains the execution of untrusted code to a space-time sandbox. In other words, the untrusted context executes only for a fixed amount of time with fixed memory bounds. This microarchitecture employs GLIFT to provably show strict information flow isolation is provided between different contexts. Furthermore, GLIFT is shown to be effective at identifying timing channels in bus protocols such as I^2C and USB [22]. In this work, GLIFT is used to show how timing channels can leak information between I/O devices even if they are abiding by the protocol. The protocol is modified to include a time multiplexed behavior, which provably isolates information between devices communicating on the shared medium. These previous efforts are combined and expanded to form a complete processor running a microkernel [23]. Here a configurable architectural skeleton couples the low level hardware implementation with a microkernel. This system uses a static analysis technique called star-logic which is used to verify, at design time, that the concrete hardware implementation and any software running upon it will be free of unintended information flows.

To track information flows using GLIFT, a fundamental task is to generate GLIFT logic, which is used for label propagation. Since GLIFT accounts for information flows at a fine granularity, i.e., bit level, the GLIFT logic tends to be substantially larger than the circuit that is being monitored. In addition, as previous work has shown [24], GLIFT logic generated using certain methods has potential losses in precision. Such imprecise GLIFT logic contains false positives indicating nonexistent unintended information flows. Therefore, it is important to understand the theoretical aspects behind GLIFT logic generation, which is essential to design complexity and precision control.

The goal of this paper is to formally prove that generating precise GLIFT logic is NP-hard. It also presents several algorithms for generating GLIFT logic and provides a complexity analysis of each. Specifically, our main contributions are:

1) providing a formal proof on the NP-completeness of the precise GLIFT logic generation problem;
2) proposing several new GLIFT logic generation algorithms, namely zero-one, SOP-POS, BDD-MUX and reconvergent fanout region reconstruction and giving a formal analysis on their complexity;
3) demonstrating the practical computational complexity of different GLIFT logic generation algorithms using *IWLS* benchmarks.

The remainder of this paper is organized as follows: Section II first introduces the basics of information flow security and fundamentals of GLIFT; it then discusses the use of GLIFT for secure hardware design. We define some related concepts in Section III. Section IV performs a formal proof on the NP-completeness of generating precise GLIFT logic. In Section V, we present various algorithms for GLIFT logic generation and formally prove their complexity. Section VI provides runtime results of different GLIFT logic generation algorithms on several *IWLS* benchmarks. We conclude in Section VII.

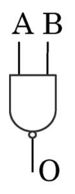
II. Preliminaries

This section introduces the basics of information flow security, describes the fundamentals of GLIFT, and shows how to use GLIFT to create a secure hardware design flow. It then covers the GLIFT logic generation problem including the potential for imprecision in GLIFT logic.

A. Basics of Information Flows

In digital systems, information flows are categorized into *explicit* and *implicit* flows. Explicit flows appear with the direct movement of data. Typical examples of explicit flows can be found in assignment expressions, where information associated with the source operand will flow to the destination operand or in bus communications, where information contained in data packets is transmitted from the sender to receiver. As its name suggests, explicit flows are easy to capture since they are dependent on direct data movements. More subtle implicit flows are caused by nondeterministic behaviors of the system, such as conditional branches or nondeterministic latency. As an example, the timing difference between a cache hit and miss creates a timing channel which has been shown to leak secret keys [16]. Implicit flows, especially hardware specific timing channels, are more difficult to detect and eliminate.

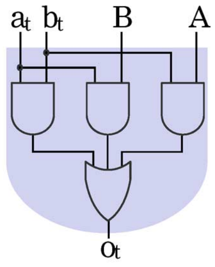
A secure system must be designed with careful consideration of both explicit and implicit flows and effective measures taken to prevent unintended flows of information. The most common technique for implementing information flow control is information flow tracking. While information flows appear in various forms at different abstraction levels of the system, GLIFT



(a)

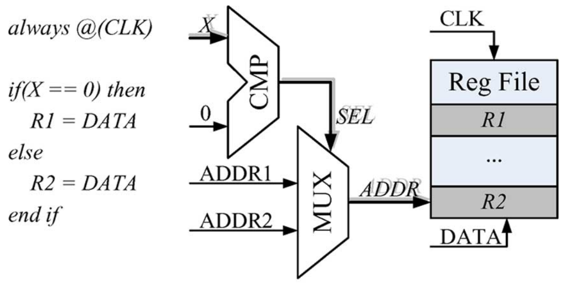
#	A	B	a_t	b_t	O	o_t
1:	**0**	**0**	**0**	**1**	**1**	**0**
2:	0	0	1	0	1	0
3:	0	1	0	1	1	0
4:	**0**	**1**	**1**	**0**	**1**	**1**
5:	1	0	0	1	1	1
6:	1	0	1	0	1	0
7:	1	1	0	1	0	1
8:	1	1	1	0	0	1

(b)



(c)

Fig. 1. (a) Two-input NAND gate. (b) Partial truth table of NAND-2 with taint information. (c) Corresponding GLIFT logic of NAND-2 is $Ab_t + Ba_t + a_tb_t$.



(a) (b)

Fig. 2. Information flows in a conditional branch. (a) Conditional branch. (b) Gate level implementation of conditional branch.

unifies the concepts of explicit flows, implicit flows, and timing channels from the level of Boolean gates. It provides a foundation for the development of secure systems by allowing hardware designers to reason about these flows. This in turn can be used to ensure private keys are never leaked (for secrecy), and that untrusted information will not be used in the making of critical decisions (for safety and fault tolerance) nor in determining the schedule (real-time). GLIFT provides a foundation for the development of secure systems by allowing hardware designers to reason about these flows.

B. Fundamentals of GLIFT

In IFT, data is assigned a label indicating its trustworthiness or security level. For example, data from an open system should be marked as untrusted and the private key used for data encryption should be labeled as secret. This label is then monitored while it propagates through the system to prevent unintended flows of information. We denote data that we wish to track as *tainted*. For example, if the integrity of data is required, the data is marked as tainted and GLIFT logic can be used to monitor if this tainted information affects critical system components. Or when the confidentiality of data is concerned, secret data is labeled as tainted and monitored to see if it ever flows to unclassified resources.

Without loss of generality, we define a bit as *tainted* when its tracking logic is true ("1") and *untainted* when its tracking logic is false ("0"). Taint is propagated from an input to the output of a Boolean function if the input has an influence on the output. As an example, consider the two-input NAND gate (NAND-2) in Fig. 1(a). Fig. 1(b) shows a partial truth table of NAND-2 with taint information, where a_t, b_t and o_t denote the taint of A, B and O, respectively. When both inputs of NAND-2 are tainted, the output will surely be tainted. Similarly, when both inputs are untainted, the output will be untainted. These obvious cases are excluded from the truth table so that we can focus on the more subtle ones in which only one input is tainted.

For a better understanding of taint propagation, consider the first row $(A = 0, B = 0, a_t = 0, b_t = 1)$ in Fig. 1(b). The tainted input B does not have an influence on the output since the output is dominated by A in this case. Thus, the output should be marked as untainted $(o_t = 0)$. Now consider row 4 $(A = 0, B = 1, a_t = 1, b_t = 0)$. The tainted input A has an influence on the output since a change in the output can be observed by changing the value of A. In this case, tainted information from input A flows to the output and the output should be marked as tainted $(o_t = 1)$. Once the GLIFT logic for NAND-2 derived from a full truth table is simplified, the resulting logic is shown in Fig. 1(c).

From the NAND-2 example, we can discover that taint is propagated from the tainted input to the output when and only when at least one tainted input affects the output. This is the basic rule used by GLIFT for taint propagation. It is different from previous conservative IFT methods that typically mark the output as tainted whenever there is any tainted input.

GLIFT also captures implicit flows at the level of primitive gates. Fig. 2 shows how all information flows in a conditional branch are made explicit in the gate level implementation.

Let the conditional variable X be tainted. If $X = 0$, $R1$ should clearly be marked as tainted ($R1$ is evaluated upon the decision made on X). Actually, even if $X \neq 0$, $R1$ should also be marked as tainted because the value of $R1$ is still dependent on X (by observing if there is a change in the value of $R1$, we can learn some information about X). Similarly, $R2$ should be marked as tainted regardless of the value of X. Such analysis is reflected in the gate level implementation of the conditional branch. The output of the comparator is used as the select line of the multiplexer, which chooses the destination register address. In this process, tainted information contained in X will flow through the select signal SEL to ADDR and eventually to both $R1$ and $R2$. GLIFT logic will indicate such implicit flows by marking the target registers as tainted. Further, GLIFT is able to capture changes in register-to-register timing since it monitors information flows at such a low abstraction level.

C. Employing GLIFT in Secure Hardware Design

GLIFT is able to account for all logic flows including explicit flows, implicit flows and timing channels. It lays a solid foundation for information flow control. Upon this foundation, verifiably secure bit-tight components, information contained architectures, and applications can be built. In practice, there are two methods for employing GLIFT logic, which we denote as static and dynamic. In a static scenario, GLIFT is used to test [22], [25] or verify [23] if the system complies with predefined information flow policies. This is done completely at design time, i.e., there is no need to physically instantiate the GLIFT logic after testing or verification is completed. Fig. 3 illustrates the use of GLIFT for static testing and verification.

Static testing/verification works as follows. The system is described in a hardware description language (HDL). Once the

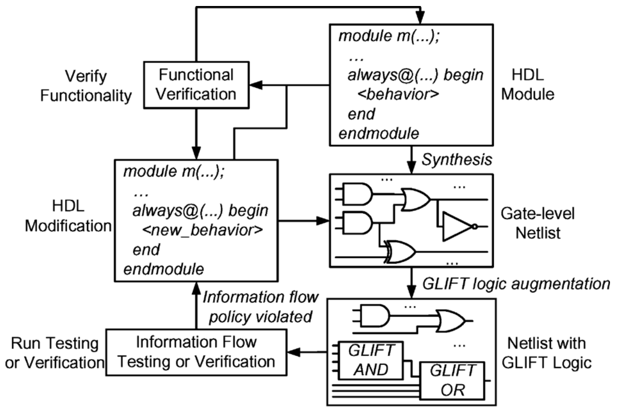


Fig. 3. Employing GLIFT for static testing or verification.

digital circuit under design has passed functional verification, it is synthesized into gate-level netlist using standard hardware synthesis tools. Then each gate in the netlist is augmented with GLIFT logic. After that, testing or verification scenarios are run to check if the design potentially violates any predefined information flow policies. If some information flow policy is violated, the design is modified and reverified until all information flow security properties are guaranteed. In the static testing or verification application scenario, the tester is responsible for specifying what data to track. For example, the tester sets certain inputs to tainted and observes if security-critical portions of the design are affected by these tainted inputs. Depending on the security scenario, a large number of information flow testing or verification scenarios may need to be run by the tester to ensure that the design is secure.

In a dynamic application scenario [20], [21], GLIFT logic is physically instantiated alongside the original hardware to allow run-time checking of security properties. In this case, system designers specify the secrecy or trustworthiness for data originating from different sources. As an example, data coming from an open network environment is labeled as tainted to indicate that it is untrusted while data from a secure separation kernel will be marked as untainted. The GLIFT logic takes both the data and their labels and performs IFT at run time. The output label will be checked to see if any information flow security policy is violated. If a violation is detected, an exception will be generated and system recovery operations will be launched. The dynamic application scheme allows the monitoring of more realistic execution patterns since some run-time features are hard to predict during design time. However, the area/delay overhead of GLIFT logic is significant. Therefore, GLIFT logic should be reserved for critical portions which require a strict information flow security guarantee.

We have demonstrated the use of GLIFT in secure hardware design as both a static testing/verification technique and a dynamic IFT approach. In [22], GLIFT is used to test information flows in bus protocols for peripheral communication with novel techniques proposed to eliminate timing channels. Other work addresses security issues in reconfigurable devices that use mix-trusted IP cores from different vendors [25]. GLIFT is used to verify that there is no unintended interaction between different modules that may violate security policies such as noninterference. In [23], a static analysis technique called star-logic is used to verify the concrete hardware implementation with partial software specification to be free of unintended information flows. The information flow properties of the entire design are statically verified all the way down the gate-level implementation using GLIFT. GLIFT has also been used as a dynamic IFT approach to build an information flow tracking microprocessor that monitors all unintended flows emanating from untrusted inputs [20]. An improved architecture [21] allows regions of execution to be tightly quarantined and their side effects to be tightly bounded, which enables mix-trust computations using shared resources through secure context switch. This new architecture also employs GLIFT to monitor information flows at run time.

In both the static and dynamic scenarios, the process of generating the GLIFT logic plays an important role. Generating low overhead and precise GLIFT logic reduces the testing/verification time in the static scenario and increases the performance of the overall system in the dynamic scenario.

D. GLIFT Logic Generation

The GLIFT logic performs label propagation to calculate the taint for each of the outputs of the original Boolean function. The inputs to the GLIFT logic are the set of inputs of the original Boolean function and the taint label set (containing the taint for each of these original inputs). Therefore, the GLIFT logic corresponding to an n-input Boolean function has $2n$ inputs and thus can be significantly more complex than the original Boolean function.

GLIFT logic exhibits various levels of precision depending on the algorithm used to generate it [24]. Preciseness is defined as indicating an information flow occurred *iff* an input *affects* the output. If information does not flow from the input to the output, yet the GLIFT logic indicates a flow, it is said to be imprecise, i.e., the GLIFT logic contains a false positive. Note that a false negative (not indicating that a flow occurred) is unacceptable in the context of IFT while a false positive (indicating that a nonexistent flow occurred) is safe but overly conservative. Excessive false positives make it difficult to determine whether or not the indicated flow is in fact harmful. Careful generation of GLIFT logic can greatly reduce (if not eliminate) these false positives.

For a more concrete understanding, consider a two-input multiplexer (MUX-2) which selects one of its inputs A and B for output according to select line S. Its Boolean function is $f = SA + \bar{S}B$. It can be implemented using two AND gates, an OR gate and an inverter. When the GLIFT logic for MUX-2 is generated by discretely instantiating tracking logic for these primitive gates,[1] impreciseness arises [24]. Specifically, an additional and unnecessary term ABs_t is introduced to the GLIFT logic. This term indicates that when A and B are both untainted "1" and the select line S is tainted, the output will be tainted. Actually, in this case, the output will be untainted "1" regardless of the status of S since both inputs are equally untainted "1". In other words, the GLIFT logic denotes a false positive.

[1]We call this the constructive algorithm as we describe in Section V-B.

TABLE I
MINTERM COUNTS OF GLIFT LOGIC FUNCTIONS OF 4-BIT ADDER GENERATED BY PRECISE AND IMPRECISE METHODS

Method	sum[0]	sum[1]	sum[2]	sum[3]	cout
Precise	229376	241664	246272	248000	208160
Imprecise	229376	245760	251648	250656	227864
F. P.%	0.00%	1.69%	2.18%	1.07%	9.47%

Now consider a 4-bit adder. Table I shows the number of minterms in the GLIFT logic functions generated using both precise and imprecise methods. We can see that the number of minterms for the imprecise method is more than or equal to that for the precise one. Such additional minterms are false positives. The percentage data in Table I show a statistic of false positives introduced by imprecise GLIFT logic generation method. As an example, the imprecise GLIFT logic for output *cout* contains as high as 9.47% false positives, which frequently indicate nonexistent tainted information flows, i.e., some information flow security policy has been violated when actually not.

With an understanding of the potential difficulties in GLIFT logic generation, this paper formally proves that generating precise GLIFT logic is an NP-hard problem and proposes several more scalable precise GLIFT logic generation algorithms. Before that, some basic concepts are formally defined.

III. TERMS AND DEFINITIONS

In this section, we provide definitions for Boolean function, binary decision diagram, reconvergent fanout and other related concepts. We do our best to use conventional notations and specifically follow the notations used in [26]–[28].

Let $\{0,1\}^n$ be a Boolean space. A *completely specified* **Boolean function** with n variables $x_1, x_2, \ldots, x_n$ is defined as a mapping: $f : \{0,1\}^n \rightarrow \{0,1\}$. For a Boolean function of n variables, a product term in which each of the n variables appears once (in either its complemented or uncomplemented form) is called a **minterm**. The **on-set** of a Boolean function is the set of minterms for which the function has value "1". The **off-set** is the set of minterms for which the function has value "0".

Definition 1 (Implicant): an **implicant** is a "covering" of one or more minterms in the on-set of a Boolean function.

Definition 2 (Prime Implicant): a **prime implicant** is an implicant that cannot be covered by a more general one.

Definition 3 (Complete Sum): the **complete sum** is the sum of all the prime implicants of a Boolean function.

Definition 4 (Static Hazard): a **static hazard** is the situation where, when one input changes, the output incorrectly changes momentarily before stabilizing to the correct value. This is typically due to different input to output paths with variations in delay. There are two types of static hazards.

1) *Static-1 Hazard*: the output is initially "1" and after an input change, the output momentarily changes to "0" before stabilizing to "1".
2) *Static-0 Hazard*: the output is initially "0" and after an input change, the output momentarily changes to "1" before stabilizing to "0".

Definition 5 [Binary Decision Diagram (BDD)]: A **BDD** is a rooted, directed acyclic graph with vertex set V containing two types of vertices. A **nonterminal** vertex v has as attributes a pointer index $\mathrm{index}(v) \in \{1, \ldots, n\}$ to a decision variable in the input variable set $\{x_1, x_2, \ldots, x_n\}$ and two children $\mathrm{low}(v), \mathrm{high}(v) \in V$. A **terminal** vertex v has as attribute a value $\mathrm{value}(v) \in \{0,1\}$.

Definition 6 (Reduced BDD): A **reduced BDD** is one that meets the following additional properties.

1) When traversing any path from a terminal vertex to the root vertex we encounter each decision variable at most once.
2) $\mathrm{low}(v) \neq \mathrm{high}(v)$ for any vertex v and no two subgraphs in the BDD are isomorphic.

Definition 7 [Reduced Ordered BDD (ROBDD)]: A canonical form called a **ROBDD** if the following restrictions are imposed: for any vertices v, $\mathrm{low}(v)$ and $\mathrm{high}(v)$ such that no vertex is terminal, we must have $\mathrm{index}(v) < \mathrm{index}(\mathrm{low}(v))$ and $\mathrm{index}(v) < \mathrm{index}(\mathrm{high}(v))$.

Definition 8 [Reduced Free (BDD)]: A **reduced free BDD** is a reduced BDD where no strict variable ordering is required on the BDD. In other words, different paths may have different variable ordering as long as each variable is encountered at most once along any path.

In our successive discussions, when talking about BDDs, we refer to reduced-ordered or free BDDs unless explicitly specified.

Definition 9 (Reconvergent Fanout): If there are two or more disjoint paths between a stem A (i.e., an input line or a logic gate output) and a gate B, then A is a **reconvergent fanout stem** (or simply reconvergent fanout), and B is a **reconvergence gate** of stem A.

Definition 10 (Reconvergent Fanout Region): The **stem region** (or reconvergent fanout region) of a reconvergent fanout stem A is composed of all the circuit nodes that satisfy the following conditions.

1) They are reached by the reconvergent fanout A.
2) They reach a reconvergence gate of stem A.

When talking about reconvergent fanout regions, we usually mean those where signal fanout and reconvergence happen explicitly. However, reconvergence may also appear in a single node containing a sum-of-product (SOP) formula. Both cases may affect the precision of the GLIFT logic and thus need to be considered. We identify the formal case as a global reconvergent fanout region and the latter as a local one.

For a better understanding, consider some examples of reconvergent fanout regions as shown in Fig. 4, where the inputs are at the bottom, the outputs are at the top, and all the numbered circles containing a Programmable Logic Array (PLA) table are internal nodes. Here, PLA tables are used to specify the functionality of internal nodes. In Fig. 4(a), there is a global reconvergent fanout region consisting of nodes $n4$, $n5$ and $n6$. In Fig. 4(b), nodes $n5$, $n6$ and $n7$ compose another global reconvergent fanout region; node $n5$ contains a local reconvergent fanout region.

Now that we have defined the basic concepts, the following section performs a formal proof on the complexity of the precise GLIFT logic generation problem.

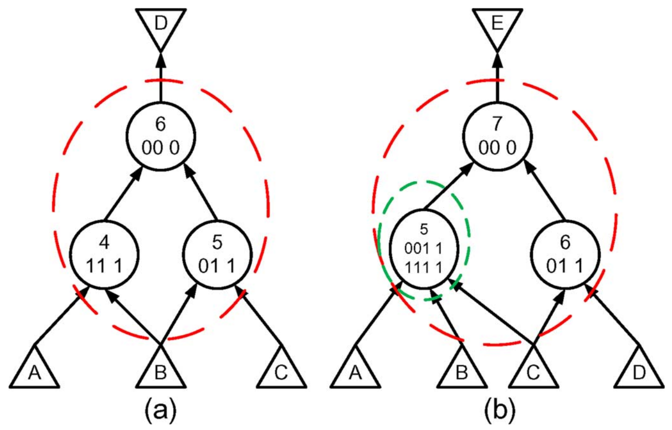


Fig. 4. Examples of reconvergent fanout regions. (a) Global reconvergent fanout region. (b) Design contains a global reconvergent fanout region and a local one.

IV. Complexity of Generating Precise GLIFT Logic

GLIFT logic augmentation is a challenging task due to its inherent complexity and possible losses of precision. In this section, we formally prove that generating precise GLIFT logic is NP-complete. In the discussion, we consider n-input Boolean functions $f(x_1, x_2, \ldots, x_n)$, where $x_1, x_2, \ldots, x_n$ are the inputs. Before performing the NP-completeness proof, we need to discuss the condition for the existence of nonconstant GLIFT logic as stated and proved in Theorem 1.

Theorem 1: A Boolean function has nonconstant GLIFT logic if the function is satisfiable and not a tautology.

Proof: If a Boolean function is not satisfiable, its value will be constant "0". In this case, no tainted input can affect the output. The GLIFT logic is independent of the tainted inputs and thus will be constantly "0". Similarly, if a Boolean function is a tautology, its GLIFT logic will be simply constant "0" as well since the output of the functions is constant "1" and can never be affected by a tainted input.

When a Boolean function is satisfiable and not a tautology, there should exist input patterns which will evaluate the Boolean function to be logic "1" and "0", respectively. In other words, tainted inputs may have a chance to affect the output. In this case, the GLIFT logic will be dependent on the tainted inputs and thus nonconstant. ■

By Theorem 1, the process of determining if a given Boolean function has nonconstant GLIFT logic can be difficult because it requires solving the NP-complete Boolean satisfiability and nontautology problems. This may provide us some initial understanding on the complexity of the GLIFT logic generation problem.

Since the NP-completeness proofs must be formalized for decision problems [29], we begin our proof by describing a decision problem related to GLIFT logic generation. Specifically, we define the *taint propagation* problem as follows:

Problem: Taint Propagation.

Instance: An n-input Boolean function f and a single tainted input x_i.

Decision Problem: Is there an input pattern that propagates the value of x_i to the output?

Now we formally prove the taint propagation problem to be NP-complete, which is stated and proved in Theorem 2.

Theorem 2: Taint Propagation is NP-complete.

Proof: 1) $\text{Taint Propagation} \in \text{NP}$.

Taint is propagated from the tainted input to the output when and only when the tainted input has an influence on the output. In other words, taint is propagated from x_i to the output when there is an input pattern $(x_1, x_2, \ldots, x_i, \ldots, x_n)$ such that:

$$f(x_1, x_2, \ldots, x_i, \ldots, x_n) \neq f(x_1, x_2, \ldots, \overline{x_i}, \ldots, x_n). \tag{1}$$

$\text{Taint propagation} \in \text{NP}$ since a nondeterministic algorithm needs only to guess a truth assignment for inputs $x_1, x_2, \ldots, x_n$ and verify in polynomial time if this input pattern propagates taint to the output by checking if (1) holds.

2) Fault Detection α Taint Propagation.

The problem of determining if a fault in an input line x_i is detectable by I/O experiments is polynomially complete [30]. We transform this NP-complete fault detection problem to taint propagation.

Given a Boolean function $f(x_1, x_2, \ldots, x_n)$ in 3-DNF (Disjunctive Normal Form), we construct a circuit C with inputs $x_1, x_2, \ldots, x_n$, such that $f(x_1, x_2, \ldots, x_n) = C(x_1, x_2, \ldots, x_n)$. This construction is a direct implementation of f and thus can be completed in polynomial time. Let a certain input x_i be tainted in the Boolean function f. We manually introduce a fault in the input line x_i of circuit C. That is, taint is propagated from x_i to the output *iff* (1) holds for some input pattern $(x_1, x_2, \ldots, x_n)$. Further, the fault in input line x_i will be detectable *iff* the same condition is satisfied.

Thus, taint propagates from x_i to the output *iff* the fault in input line x_i will be detectable. In other words, fault detection transforms to taint propagation. ■

Generating precise GLIFT logic is a search problem. As such, we defined the corresponding search problem of the taint propagation decision problem as:

Problem: Taint Propagation.

Instance: An n-input Boolean function f and a single tainted input x_i.

Search Problem: Find an input pattern that propagates the value of x_i to the output, or else output $\perp$.

The search problem of a NP-complete problem is no easier than the decision problem itself [29]. This is due to the fact that the solution to the search problem provides an answer to the decision problem. Usually, a solution to the search problem can be found by solving the corresponding decision problem a polynomial number of times. For the taint propagation problem, a satisfied input pattern can be found by solving its corresponding decision version n times if there is one. Once a solution to the taint propagation search problem is found, a minterm can be added to the GLIFT logic to track tainted information flowing from input x_i to the output under this input pattern. Thus, precisely determining which minterms with a single tainted input should be added into the GLIFT logic requires solving the NP-complete decision and this NP-hard search problem and thus is even harder.

Apart from single tainted input cases, there are usually minterms with multiple tainted inputs in the GLIFT logic as well. We will show that precisely determining which minterms with multiple tainted inputs should be added into the GLIFT logic is equally hard. Before that, it is necessary to prove the following lemma.

Lemma 1: The output of a Boolean function will be tainted when at least one tainted input propagates to the output.

Proof: By the definition of taint, the output of a Boolean function will be tainted when at least one tainted input has an effect on the output. In other words, the output will be tainted when at least one tainted input propagates to the output. ■

Theorem 3: Finding an input pattern that propagates taint from any of the m tainted inputs $x_{i1}, x_{i2}, \ldots, x_{im}$ to the output is NP-hard.

Proof: By Lemma 1, one needs to solve the taint propagation decision problem m times to determine if any tainted input is able to propagate to the output in the worst case. Once there exists a tainted input that propagates to the output, an input pattern can be found by solving the taint propagation decision problem another n times. Thus, finding an input pattern that propagates taint from any of the m tainted inputs to the output is determined by solving the taint propagation decision problem a polynomial number of times. Since the taint propagation decision problem is NP-complete, finding an input pattern that propagates taint from any of the m tainted inputs to the output is NP-hard. ■

Once an input pattern that propagates taint from any of the m tainted inputs $x_{i1}, x_{i2}, \ldots, x_{im}$ to the output is found, a minterm is added to the GLIFT logic with these m inputs marked as tainted. Therefore, precisely determining which minterms with multiple tainted inputs should be added into the GLIFT logic requires solving an NP-complete decision and an NP-hard search problem and thus is even more complex. Since we showed that precisely determining which minterms with either single or multiple tainted input(s) should be added into the GLIFT logic needs to solve NP-complete problems, precise GLIFT logic generation (i.e., the corresponding optimization problem of taint propagation which requires finding all possible solutions) is also a hard problem.

Taint propagation, the fundamental problem of GLIFT logic generation, closely relates to several well-known problems in the switching circuit theories including Boolean satisfiability, nontautology, fault detection, observability and automatic test pattern generation (ATPG). Specifically, if taint is able to propagate from any input to the output, the Boolean function should be satisfiable and also not a tautology. If taint cannot be propagated from any input to the output, a single evaluation step will determine if the Boolean function is a tautology or unsatisfiable. Fault detection and observability are both concerned with the existence of an input pattern that propagates the value of some signal to an observation point. It is directly related to taint propagation. ATPG takes a step further by finding effective test vectors that solve the fault detection or observability problem if any exists. These known problems are all concerned about the propagation of values through Boolean functions or circuits. In addition, all these problems have been proved to be NP-complete [29]–[31], which provides a good insight into the complexity of our taint propagation and GLIFT logic generation problems.

V. GLIFT Logic Generation Algorithms

This section introduces various GLIFT logic generation algorithms with a formal analysis on their computational complexity and precision.

A. Minterm Enumeration Algorithms

The brute force algorithm, which we formalized in previous work [24], is a minterm enumeration algorithm based upon the definition of information flow. It works by changing the inputs to a given Boolean function and observing what combinations can cause a difference in the output. For any combination that causes a change in the output, a minterm is added to the GLIFT logic with the changed logic variables(s) marked as tainted and unchanged variable(s) marked as untainted.

As an example, consider the NAND-2 gate (see Fig. 1). Assume the initial inputs under consideration are $A = 0$ and $B = 1$. By changing the value of A to "1", the output will change from "1" to "0". Thus, a difference in the output is observed and the minterm $\bar{A}Ba_t\overline{b_t}$ should be added to the GLIFT logic. This minterm monitors the flow of tainted information from input A to the output when B is untainted and logically "1". Then let $A = 1$ and $B = 1$. By changing the value of A to "0", the output will change from "0" to "1". Thus, a change in the output is observed and the minterm $ABa_t\overline{b_t}$ should also be added to the GLIFT logic. Note that the two minterms $\bar{A}Ba_t\overline{b_t}$ and $ABa_t\overline{b_t}$ can be combined and reduced to the implicant $Ba_t\overline{b_t}$ resulting in simplified GLIFT logic.

The GLIFT logic generated using the brute force algorithm correctly tracks all flows of information assuming all input combinations are analyzed. Furthermore, the brute force algorithm only accounts for the actual information flows, i.e., it is precise for GLIFT logic generation. However, this algorithm has high computational complexity because every single input combination must be checked in order to accurately determine which minterms should be added to the GLIFT logic.

Theorem 4: The complexity of the brute force algorithm is $O(2^{2n})$.

Proof: For an n-input Boolean function, there are a total of 2^n minterms; generating all these minterms takes 2^n steps. For every minterm, each of the remaining $2^n - 1$ minterms must be checked to see if differences in the inputs lead to a change in the output. This is redundant since each pair of terms only needs to be checked once. As such, the total number of checking operations actually needed is $2^n \cdot (2^n - 1)/2$. Thus, the algorithm will complete in $2^n + 2^n \cdot (2^n - 1)/2$, i.e., $2^{n-1} + 2^{2n-1}$ steps. Therefore, the computational complexity of the brute force algorithm is $O(2^{2n})$. ■

An improvement to this brute force algorithm is the zero-one algorithm, which enumerates minterms in a slightly more efficient way. This improved algorithm essentially performs mapping from the on-set of a Boolean function to its off-set. Each mapping operation will result in an implicant containing the taint(s) of changed variable(s) and the literal(s) of unchanged variable(s). This is based upon the inherent property of GLIFT that the value of a taint variable can be ignored in taint propagation and the complement of taint never appears in simplified GLIFT logic [32]. As a result, the mapping operations will produce cubes rather than minterms.

For a better understanding, once again consider the NAND-2 example. The on-set of NAND-2 is $S_{\text{on}} = \{00, 01, 10\}$ while the off-set is $S_{\text{off}} = \{11\}$ (assume a variable ordering of AB). When mapping from "00" to "11", the implicant $a_t b_t$ should be added to the GLIFT logic since both A and B are tainted, their values can be ignored; mapping from "01" to "11" will result in the implicant Ba_t since A is tainted, its value can be ignored and now that B is untainted, the complement of its taint, i.e., $\overline{b_t}$, can be eliminated. According to the definition of information flow, the GLIFT logic generated using the zero-one algorithm is also precise.

Theorem 5: The upper bound on complexity of the zero-one algorithm is $2^n + 2^{2n-2}$.

Proof: Consider an unsimplified Boolean function, whose on- and off-sets both consist of minterms. For an n-input function, there are $C \cdot 2^n (0 < C < 1)$ minterms in its on-set and $(1-C) \cdot 2^n$ minterms in its off-set. The computation time needed for minterm generation is 2^n.

Each single map operation from the on-set to the off-set corresponds to an implicant in the GLIFT logic. Thus, the zero-one algorithm can complete in $C(1-C) \cdot 2^{2n}$ steps. Since $C(1-C) \cdot 2^{2n}$ reaches maximum when $C = 1/2$, the computational complexity of the zero-one algorithm is bounded by $2^n + 2^{2n-2}$. ■

In practice, if minterms in the on- and off-sets of a Boolean function are combined to more general implicants, the number of elements in these two sets will decrease dramatically. In this case, the computation time of the zero-one algorithm will be reduced significantly. Additionally, the resulting implicants from each mapping operation will yield even larger cubes. However, according to [33], the number of product terms in the SOP representation of an n-input Boolean function is bounded by 2^{n-1}. There are known functions with exactly 2^{n-1} product terms in both its on- and off-sets, e.g., n-input *XOR* and *NXOR*. In this case, the complexity of the zero-one algorithm will reach $O(2^n + 2^{2n-2})$. Thus, the complexity of minterm enumeration algorithms is generally on the order of $O(2^{2n})$, which makes them inefficient for processing large Boolean functions. However, these two algorithms are directly based on the definition of information flow and thus provide a good basis for understanding how to generate GLIFT logic.

The minterm enumeration algorithms perform exhaustive search in the on- and off-sets of a Boolean function, and the problem will become intractable as the number of inputs increases. Our experiments show that these minterm enumeration algorithms have difficulty processing even moderately complex designs. Advances in switching circuit theory, e.g., Boolean difference and SAT based observability don't care analysis, may provide more scalable solutions than minterm enumeration. These methods lead to generalized solutions that yield cubes instead of minterms, which could be dramatically more scalable and in line with logic synthesis techniques. As an example, consider a Boolean function f with a tainted input x_i. One can construct another Boolean function f_{x_i} with only x_i inverted in the Boolean formula of f. By applying an all-solutions SAT solver on $f \oplus f_{x_i}$, one can precisely obtain all input patterns that propagates tainted information from x_i to the output if any exists. Although these methods are more scalable, their efficiency is restricted to single tainted input analysis. Consider a multiple tainted input case with m tainted inputs; these tainted inputs do not necessarily all change their values in a single analysis. Instead, one needs to assume i tainted inputs indeed change their values for a certain run. Since the total number of variable combinations is exponential, i.e., 2^n, the number of runs will grow exponentially in a single multiple tainted input analysis.

With an understanding of the complexity of minterm enumeration algorithms, the following sections focus on more efficient solutions for GLIFT logic generation using techniques other than exhaustive search.

B. Constructive Algorithm

The constructive algorithm provides a less computationally complex approach to GLIFT logic generation. However, it is not guaranteed to provide a precise GLIFT function [24]. This algorithm maintains a GLIFT library that contains the tracking logic for gate primitives such as AND, OR and NOT T. Given a Boolean function, gate primitives in its logic equation are augmented with tracking logic from the GLIFT library discretely, which is similar to technology mapping.

Let g denote the number of gates in the circuit representation of a Boolean function. Then, the complexity of the constructive algorithm will be polynomial to g. This is formally stated and proved as Theorem 6.

Theorem 6: The complexity of the constructive algorithm is $O(g)$.

Proof: In the constructive algorithm, each gate in a Boolean function is augmented with tracking logic through a constant time mapping operation. Therefore, the time for GLIFT logic generation is $C \cdot g$, where C is the constant associated with tracking logic mapping for an individual gate. Thus, the complexity of the constructive algorithm is $O(g)$. ■

While the computation time of the constructive algorithm is linear to the number of gate primitives in a Boolean function, GLIFT logic generated using this algorithm can be imprecise. Such imprecision is caused by one-variable switches (multiple variable switches do not cause imprecision) [24]. Switching circuit theories in static hazards [34], [35] and reconvergent fanouts [28] address such variable switch activities and thus both provide a good insight to the impreciseness of the constructive algorithm. The following sections present solutions to this imprecision problem from these two viewpoints.

C. Complete Sum Algorithm

As mentioned, the impreciseness of the constructive algorithm is caused by the correlation between a variable and its complement, i.e., a single variable switch. Switching circuit theory observes that such single variable switches result in static hazards [34]. It has been proven that a logic circuit containing all its prime implicants is free of all static hazards [35]. Thus, GLIFT logic generated using the constructive algorithm from a Boolean function in its complete sum form will be precise. We call this precise GLIFT logic generation approach the complete sum algorithm and have formally proven it to be precise for GLIFT logic generation [24].

However, the complete sum algorithm is computationally expensive since the generation of just one prime implicant from a normalized Boolean formula is NP-hard [36]. Further, the total number of prime implicants of a Boolean function is generally exponential to the number of inputs of that function. For an n-input Boolean function, the maximum number of prime implicants approaches $3^n/\sqrt{n}$ [37]. There are known functions with a total number of $3^n/n$ prime implicants. Thus, the complete sum algorithm is inherently exponential, which is formalized as proved in Theorem 7.

Theorem 7: The complexity of the complete sum algorithm is $O(2^n + g)$.

Proof: The complete sum algorithm is complete in two steps. First, all prime implicants of a given Boolean function are derived. This step is considered to be inherently of exponential complexity, regardless of the representation of a Boolean function [38]. The computational complexity of known algorithms, for example, the Quine's algorithm [39] is $O(n!)$. Most of the other methods are $O(3^n)$ [40], [41], or some are $O(2^n)$ [38], [42]. The second step of the algorithm constructively augments tracking logic for all prime implicants, whose computation time is polynomial to the total number of prime implicants g. Thus, the complexity of the complete sum algorithm is $O(2^n + g)$. ■

For simple functions with a small number of prime implicants, there are existing tools such as *ESPRESSO* [43], which are efficient for finding all prime implicants. However, the complete sum algorithm is inherently expensive because it requires solving the NP-hard prime implicants generation problem. This provides us some further understanding on the complexity of precise GLIFT logic generation [36]. In the next section, we propose a new algorithm which requires the calculation of more general two-level representations instead of complete sum for generating precise GLIFT logic.

D. SOP-POS Algorithm

The impreciseness of the constructive algorithm is caused by static hazards. A well-known property in switching circuit theory is that a circuit in sum-of-products (SOP) representation is automatically free of static-0 hazards and a circuit in product-of-sum (POS) form is free of static-1 hazards [44]. As a consequence, the false positives in GLIFT logic generated from circuits in SOP representation using the constructive algorithm are caused by static-1 hazards while those in GLIFT logic generated from circuits in POS form constructively are caused by static-0 hazards. Further, these two imprecise GLIFT logic functions do not overlap in their false positives since the static-1 and static-0 hazards of a Boolean function never have an intersection [45]. Thus, we can generate two imprecise GLIFT logic functions from the SOP and POS representations of a Boolean function and obtain the precise GLIFT logic by performing a logic AND operation on the two imprecise ones. We call this new precise GLIFT logic generation approach the SOP-POS algorithm.

Given a Boolean function f, we denote its SOP and POS representations as f_{SOP} and f_{POS}; the GLIFT logic functions generated from them constructively are denoted as $sh(f_{\mathrm{SOP}})$ and $sh(f_{\mathrm{POS}})$, respectively.[2] Then, the precise GLIFT logic $sh(f)$ can be obtained using

$$sh(f) = sh(f_{\mathrm{SOP}}) \cdot sh(f_{\mathrm{POS}}). \tag{2}$$

For a concrete understanding, once again consider the MUX-2 example. We have

$$\begin{aligned} f_{\mathrm{SOP}} &= SA + \bar{S}B \\ f_{\mathrm{POS}} &= (\bar{S} + A)(S + B). \end{aligned} \tag{3}$$

When generated constructively, $sh(f_{\mathrm{SOP}})$ as shown in (4), contains false positive ABs_t (caused by static-1 hazard in f_{SOP})

$$\begin{aligned} sh(f_{\mathrm{SOP}}) = \bar{S}b_t + Sa_t + \bar{A}Bs_t + A\bar{B}s_t \\ + a_t s_t + b_t s_t + ABs_t. \end{aligned} \tag{4}$$

Using the constructive algorithm, we get $sh(f_{\mathrm{POS}})$, as given in (5), which contains false positive $\bar{A}\,\bar{B}s_t$ (caused by static-0 hazard in f_{POS})

$$\begin{aligned} sh(f_{\mathrm{POS}}) = \bar{S}b_t + Sa_t + \bar{A}Bs_t + A\bar{B}s_t \\ + a_t s_t + b_t s_t + \bar{A}\,\bar{B}s_t. \end{aligned} \tag{5}$$

When performing a logic AND of (4) and (5), both false positives will be reduced and the resulting GLIFT logic is shown in (6), which is equivalent to the precise GLIFT logic generated using the brute force algorithm

$$sh(f) = \bar{S}b_t + Sa_t + \bar{A}Bs_t + A\bar{B}s_t + a_t s_t + b_t s_t. \tag{6}$$

In practice, the SOP (or POS) representations of f and $\bar{f}$ are also eligible for precise GLIFT logic generation. The resulting GLIFT logic will be equally precise. This is because they do not have an intersection in their static-1 (or static-0) hazards and f and $\bar{f}$ share the same precise GLIFT logic [32].

In general, the computational complexity of calculating two PLA tables can be significantly lower than finding all prime implicants. Thus, the SOP-POS algorithm can be more efficient than the complete sum algorithm. Theorem 8 formally states and proves the complexity of the SOP-POS algorithm.

Theorem 8: The upper bound on complexity of the SOP-POS algorithm is $O(2^n + g + 1)$.

Proof: The SOP-POS algorithm generates precise GLIFT logic from two PLA (either SOP or POS) tables of a given Boolean function. For an n-input function, the complexity of computing two PLA tables has an upper bound of 2^n (the on- and off-sets of the function in the worst case). With these two PLA tables, the two imprecise GLIFT logic functions can be generated in polynomial time, which is linear to the total number of gates g in the two PLA tables. In addition, the AND operation on the two imprecise GLIFT logic functions takes constant time. Thus, the complexity of the SOP-POS algorithm is upper bounded by $O(2^n + g + 1)$. ■

The SOP-POS algorithm requires calculating more general two-level representations instead of the complete sum of a Boolean function, which makes it less expensive as compared

[2]The notation *sh* is derived from the "shadow logic", which is another term for GLIFT logic.

to the complete sum algorithm. For an n-input Boolean function f, the number of product terms in its SOP representation has an upper bound of 2^{n-1}. There are existing functions that have 2^{n-1} product terms in both f and $\bar{f}$, such as n-input XOR. The calculation of two SOP formulas takes a total of 2^n steps in this case. Thus, the complexity of the SOP-POS algorithm can reach $O(2^n + g + 1)$.

With the consistent growth in the size of integrated circuits, modern synthesis tools usually use multilevel logic network for circuit representation. Mapping of the multilevel representation of a Boolean function to two-level SOP/POS forms has scalability challenges, which makes the SOP-POS algorithm inefficient for processing large multilevel logic networks. In the following section, we consider a frequently used multilevel logic representation technique and present a more scalable precise GLIFT logic generation algorithm.

E. BDD-MUX Algorithm

Recent advances in digital circuit design has enabled the synthesis of static hazard-free multilevel logic using BDDs [27]. Now that this BDD method is able to eliminate static hazards, which are the sources of impreciseness of the constructive algorithm, it can also be used for precise GLIFT logic generation.

To generate precise GLIFT logic, a reduced ordered or free BDD is constructed from a given Boolean function. Then a multilevel logic network is derived from the BDD by replacing each BDD vertex with a two-input multiplexer (MUX-2). After that, the multiplexer network is simplified through constant propagation. There is some difference in the constant propagation step as compared to the method given by [27]. We perform constant propagation only when both inputs to a multiplexer are constants. In case only one input is a constant, we label the constant input as untainted. Finally, the multiplexer network is augmented with GLIFT logic using the constructive algorithm. As long as the GLIFT logic for the MUX-2 in the GLIFT library is precise, the resulting GLIFT logic function will be precise. We call this new precise GLIFT logic generation algorithm the BDD-MUX algorithm.

For a more concrete understanding, consider the Boolean function $f = AB + \bar{A}(\bar{B} + C)$. When generating GLIFT logic for f using the constructive algorithm directly, the resulting GLIFT function will be imprecise because there is a one-variable switch caused by subterms AB and $\bar{A}C$. To generate precise GLIFT logic using the BDD-MUX algorithm, we first construct a BDD from f as shown in Fig. 5(a). Then all the vertexes in the BDD are substituted with MUX-2, which results in a multiplexer network as shown in Fig. 5(b). The multiplexers with two constant inputs are further simplified through constant propagation. Finally, the simplified logic network shown in Fig. 5(c) is augmented with GLIFT logic constructively using the precise tracking logic for MUX-2 as given in (6).

Theorem 9: The complexity of the BDD-MUX algorithm is $O(2^n + g)$.

Proof: The BDD-MUX algorithm first constructs a reduced free or ordered BDD for a given Boolean function. For an n-input Boolean function, the computational complexity of

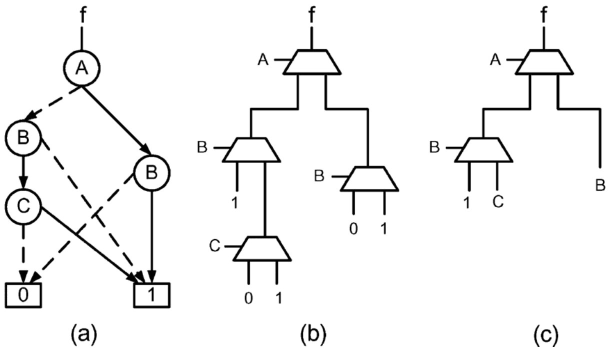


Fig. 5. (a) BDD. (b) Derived multiplexer network. (c) Simplified logic network after constant propagation.

BDD construction has an upper bound of 2^n [46]. The constructive GLIFT logic augmentation process will complete in g steps, where g is the number of multiplexers in the simplified multiplexer network. Thus, the complexity of the BDD-MUX algorithm is $O(2^n + g)$. ■

The most computationally expensive step of the algorithm lies in the construction of a reduced free or ordered BDD from a Boolean formula. Certain NP-complete problems such as SAT and nontautology can be solved in polynomial time on a reduced free or ordered BDD. Thus, the construction of a reduced free or ordered BDD is generally of exponential complexity. Fortunately, in most cases such a BDD can be constructed in a time complexity far less than $O(2^n)$. Proper variable ordering can significantly decrease both the time and space complexity of BDD construction, which makes the BDD-MUX algorithm faster than other GLIFT logic algorithms in practice as we will show in Section VI.

The BDD-MUX algorithm targets multilevel logic networks instead of two-level tabular or SOP/POS representations. The number of nodes in the reduced BDD of a Boolean function is bounded by $2^n/n$. By comparison, the number of product terms in its SOP formula is bounded by 2^{n-1}. Thus, the BDD-MUX algorithm is often more scalable than the previous precise GLIFT logic generation algorithms. In addition, there are mature BDD packages such as CUDD [47] and even existing logic synthesis tools such as *ABC* [48] that provides support for BDD maintenance.

The complete sum, SOP-POS and BDD-MUX algorithms consider the cause of impreciseness of the constructive algorithm as static hazards. We can also consider the imprecision problem from the viewpoint of reconvergent fanouts. The following section presents a precise GLIFT logic generation algorithm that deals with reconvergent fanout regions.

F. Reconvergent Fanout Region Reconstruction Algorithm

Reconvergent fanout regions have been identified as potential sources of logic hazards and race conditions in the switching circuit theory [28]. In a reconvergent fanout region, there is the possibility for the occurrence of one-variable switch (i.e., static hazard). Thus, such regions are potential sources of impreciseness of the constructive algorithm. To generate precise GLIFT logic, reconvergent fanout regions need to be locally reconstructed either as a single node in the complete sum form or a multiplexer network translated from a reduced free or ordered BDD. Finally, the fully processed logic network is augmented with GLIFT logic using the constructive algorithm. We call this new precise GLIFT logic generation approach the reconvergent fanout region reconstruction algorithm. In successive discussions, it will be also called the *RFRR* algorithm for simplicity.

A multilevel logic network can be described as a directed graph $G(V, E)$, where V is the vertex set and E is the edge set. Let $v = |V|$ and $e = |E|$. The complexity of the RFRR algorithm is determined by the number of elements in these two sets, together with the number of inputs of the Boolean function. This is formalized as Theorem 10.

Theorem 10: The complexity of the reconvergent fanout region reconstruction algorithm is $O((v+e)\cdot 2^{e-v+n}+g)$.

Proof: The computation extensive steps of the algorithm lie in the search for all reconvergent fanout regions and reconstructing them as a single node in the complete sum form or a multiplexer network translated from a reduced free or ordered BDD. In a logic network $G(V, E)$, the maximum number of global reconvergent fanout regions is $2^{e-v+1}-1$ and the number of local ones is v in the worst case. The complexity of finding one reconvergent fanout region is bounded by $O(v+e)$. Assume a reconvergent fanout region has m inputs. The complexity for reconstruction would be $O(2^m)$. Thus, for an n input Boolean function, the reconvergent fanout region construction algorithm will complete in $(v+e)\cdot(2^{e-v+n+1}+(v-1)\cdot 2^n)+g$ steps in the worst case ($m = n$ for all reconvergent fanout regions), where g is the number of gates in the fully processed logic network. Thus, the complexity of the reconvergent fanout region construction algorithm is $O((v+e)\cdot 2^{e-v+n}+g)$. ■

The RFRR algorithm has higher complexity as compared to the BDD-MUX algorithm which also targets the multilevel logic network. Theoretically, the complete sum algorithm is the extreme case of the RFRR algorithm, where a primary output is considered as a reconvergence gate. Although this algorithm is not as efficient as the BDD-MUX algorithm, it considers the imprecision problem from a different viewpoint, i.e., reconvergent fanouts which cause data correlations. This algorithm also provides a possibility to capture where impreciseness initially arises and identify portions of the circuit that need redesign to (partially) eliminate imprecision.

G. Comparison of GLIFT Logic Generation Algorithms

The computational complexity and precision of different GLIFT logic generation algorithms are summarized as shown in Table II.

From Table II, the complexity of the brute force and zero-one algorithms are on the same order, i.e., $O(2^{2n})$. However, the zero-one algorithm may take relatively less computation time since the exact upper bounds of their complexity are 2^{2n-1} and 2^{2n-2}, respectively. In addition, if general implicants are considered instead of minterms, the reduction in execution time of the zero-one algorithm will be even more significant. The complexity of the complete sum, SOP-POS and BDD-MUX algorithms are on the same order. They are one order of magnitude faster than the brute force and zero-one algorithms. However,

TABLE II
COMPUTATIONAL COMPLEXITY AND PRECISION OF DIFFERENT GLIFT LOGIC GENERATION ALGORITHMS

Algorithm	Computational Complexity	Precise
Brute Force	$O(2^{2n})$	*YES*
Zero-one	$O(2^{2n})$	*YES*
Constructive	$O(g)$	*NO*
Complete Sum	$O(2^n+g)$	*YES*
SOP-POS	$O(2^n+g+1)$	*YES*
BDD-MUX	$O(2^n+g)$	*YES*
RFRR	$O((v+e)\cdot 2^{e-v+n}+g)$	*YES*

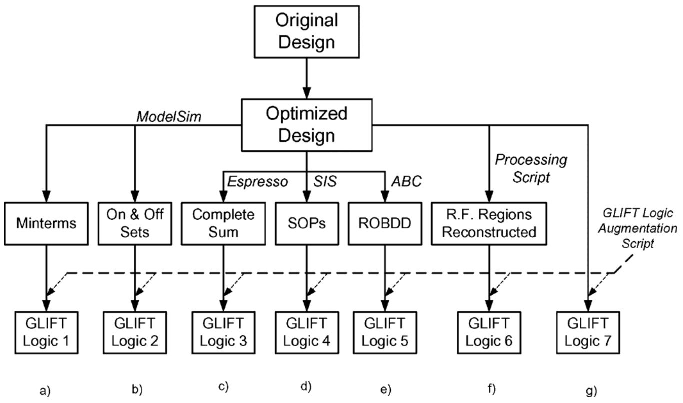


Fig. 6. Experimental flows for runtime analysis of various GLIFT logic function generation algorithms. (a) Brute force, (b) zero-one, (c) Com. Sum, (d) SOP-POS, (e) BDD-MUX, (f) RFRR, and (g) constructive.

the SOP-POS and BDD-MUX algorithms execute faster than the complete sum algorithms in most cases. This is because calculating two PLA tables or a reduced ordered or free BDD is generally faster than finding all its prime implicants. The RFRR algorithm is inherently expensive since the number of reconvergent fanout regions can be exponential and reconvergent fanout region reconstruction has exponential complexity as well. The constructive algorithm is the only method whose complexity is polynomial to the number of primitive gates in a given design. Yet it is also the only algorithm that has potential losses in precision.

VI. EXPERIMENTAL RESULTS

A. Experimental Setup

We carried out experiments on several *IWLS* benchmarks to obtain runtime results of the different GLIFT logic generation algorithms. In our experiments, the GLIFT logic functions are generated using the algorithms discussed in this paper. These algorithms use the design tools shown in Fig. 6.

The *brute force* algorithm works on a full truth table generated by the *ModelSim* simulation tool. Our GLIFT logic augmentation script checks each minterm in the truth table against the remaining minterms to see if there is a difference in the output. Whenever a difference is encountered, a minterm with the changed variables marked as tainted and unchanged variables marked as untainted is added to the GLIFT logic. The *zero-one* algorithm further partitions the truth table generated into the on- and off-sets of its Boolean function. Then our GLIFT logic augmentation script maps minterms in the on-set

TABLE III
RUNTIME OF DIFFERENT GLIFT LOGIC GENERATION ALGORITHMS. "N. AVG" IS AVERAGE RUNTIME NORMALIZED TO THAT OF CONSTRUCTIVE ALGORITHM. RESULTS ARE IN SECONDS (SEC)

Bench.	Brut.	Zero.	Con.	Com.	SOP.	BDD.	RFRR
ttt2	-	-	0.10	0.52	0.28	0.33	0.99
t481	-	680.7	0.49	0.46	0.66	0.06	21.02
alu2	11.08	2.67	0.14	0.36	0.37	0.27	2.16
alu4	3124	291.1	0.27	1.69	1.16	1.20	8.06
apex6	-	-	0.58	3.69	1.90	1.42	7.20
vda	-	-	0.37	1.44	1.24	1.78	3.09
x1	-	-	0.20	1.03	1.12	0.92	0.21
x3	-	-	0.89	3.71	1.97	1.41	7.04
x4	-	-	0.43	2.71	1.41	1.32	2.53
i5	-	-	0.19	1.56	1.17	1.00	2.23
i6	-	-	0.30	1.51	1.08	0.54	2.91
i7	-	-	0.39	1.67	1.14	0.62	3.45
i8	-	-	1.06	4.73	2.92	3.53	85.67
i9	-	-	0.34	7.85	3.03	2.83	26.39
frg2	-	-	0.59	8.37	6.74	4.96	17.47
DES	-	-	1.85	243.1	159.3	47.5	428.1
N.Avg.	-	-	1.00	14.47	9.42	5.04	36.46

to the off-set. For each mapping operation, a minterm with the unchanged variables and taints of changed variables is added to the GLIFT logic. The *complete sum* algorithm uses *ESPRESSO* [43] to find all prime implicants of the benchmark. Then the constructive algorithm operates on the complete sum representation to create the GLIFT logic. The *SOP-POS* algorithm uses *SIS* [49] to calculate the SOP representations for both the original benchmark and the complemented design. Then our script generates GLIFT logic functions for both representations. Finally, the precise GLIFT logic is obtained by performing an AND operation on the two imprecise GLIFT logic functions. The *BDD-MUX* algorithm uses *ABC* [48] to construct a ROBDD for the benchmark and translate the ROBDD to a multiplexer network. Then we use our script to create the GLIFT logic for the multiplexer network constructively. The *RFRR* algorithm uses our preprocessing script to find and reconstruct all reconvergent fanout regions either as a SOP formula in the complete sum form or as a multiplexer network translated from a ROBDD. Then we use the constructive algorithm to create the final GLIFT logic. The *constructive* algorithm processes the optimized logic circuit directly using our own GLIFT logic augmentation script and creates a potentially imprecise GLIFT logic function.

The GLIFT logic functions generated using algorithms a) to f) as denoted in Fig. 6 are all precise and thus should be logically equivalent. This is verified using the formal equivalence checking command in the *ABC* tool. The GLIFT logic function generated using flow g) may be imprecise and therefore is not necessarily equivalent to the other functions. The following section outlines the runtime results of different algorithms.

B. Runtime Results

GLIFT logic functions for several *IWLS* benchmarks are generated using the algorithms described in this paper. Their execution time is shown in Table III. We stop if an algorithm cannot complete on a benchmark within 10 hours. We are restricted in the benchmarks we could test because we used several logic synthesis tools such as *ESPRESSO* and *ABC* in our experiment. These tools have limitations on the size of the circuit they could process.

Consider the benchmark *DES* in Table III which is an implementation of the *Data Encryption Standard*. The "-" symbols indicate that the brute force and zero-one algorithms required over 10 hours to complete on this benchmark. The constructive algorithm takes only 1.85 s. However, the GLIFT logic generated by this constructive algorithm is imprecise as compared to those generated using the remaining algorithms. The complete sum, SOP-POS, BDD-MUX and RFRR algorithms require execution times of 243.1, 159.3, 47.5, and 428.1 s, respectively. The last row of Table III shows the average runtime normalized to the execution time of the constructive algorithm.

The brute force and zero-one algorithms are the most expensive since they require greater than 10 hours on many of the benchmarks. The constructive algorithm often takes the shortest time to complete while the RFRR algorithm usually needs a longer execution time. The SOP-POS and BDD-MUX algorithms typically see less computation time than the complete sum algorithm since deriving PLA tables or a BDD is typically faster than finding all prime implicants. Further, we see that the BDD-MUX algorithm is close to or faster than the SOP-POS algorithm in most cases. For certain benchmarks, the BDD-MUX algorithm executes significantly faster, such as *t481* and *DES*. The RFRR algorithm is slow because of the large number of reconvergent fanout regions and inherent complexity of reconstruction. It works quickly when few reconvergent fanout regions need to be processed such as in the *x1* example. It is necessary to point out that the runtime of different algorithms is function-specific. It depends on both the functionality (e.g., the percentage of minterms included in the on-set affects the total number of prime implicants) and description style of the benchmark.

The constructive algorithm has the lowest complexity while all the remaining algorithms are inherently expensive. However, precision of GLIFT logic is also an important factor in security critical applications. There can be various tradeoffs for system designers. In highly secure systems that require high precision, sacrifices in computational complexity need to be made and precise GLIFT logic generation algorithms should be used. While in systems where a certain amount of false positives can be tolerated, imprecise GLIFT logic can be generated using the constructive algorithm in polynomial time. The amount of precision required to adequately uphold the information flow security policy of an application remains an open problem. System designers should make the final decision how precise the GLIFT logic needs to be upon tradeoffs between security requirements and design efforts.

VII. CONCLUSION

GLIFT provides an effective approach to monitor information flows including those through hardware specific timing channels from Boolean functions. It can be integrated into the standard hardware design and verification process for eliminating unintended interactions between subsystems which may open up doors for malicious attacks. This paper presents formal proof on the NP-completeness of precise GLIFT logic generation. Several GLIFT logic generation algorithms are proposed with formal analysis on their complexity and precision. In highly secure systems where high precision is required, precise GLIFT logic generation algorithms should be used, while in applications where imprecision can be tolerated, the constructive algorithm would be more cost effective. Further, heuristic algorithms such as adding prime implicants to or selectively reconstructing reconvergent fanout regions can also be used. The algorithm can stop when a desired precision is achieved, which provides flexible tradeoffs between precision and design efforts.

Acknowledgment

The authors would like to thank the reviewers for their valuable feedback, which was of great help in improving this paper.

References

[1] "Federal aviation administration (FAA)," Special Conditions: Boeing Model 787-8 Airplane; Systems and data networks security-isolation or protection from unauthorized passenger domain systems access 2008 [Online]. Available: http://cryptome.info/faa010208.htm

[2] C. Li, A. Raghunathan, and N. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *Proc. 13th IEEE Int. Conf. e-Health Networking Applications and Services (Healthcom)*, Columbia, MO, Jun. 2011, pp. 150–156.

[3] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *Proc. IEEE Symp. Security and Privacy, 2008*, Oakland, CA, May 2008, pp. 129–142.

[4] "Common criteria," Common criteria for information technology security evaluation 2009 [Online]. Available: http://www.commoncriteriaportal.org/cc/

[5] G. Heiser, What Does cc eal6+ mean? 2008 [Online]. Available: http://www.ok-labs.com/blog/entry/what-does-cc-eal6-mean/

[6] Green Hills, The integrity real-time operating system 2010 [Online]. Available: http://www.ghs.com/products/rtos/integrity.html

[7] D. Bell and L. LaPadula, Secure computer systems: Mathematical foundations MITRE Corporation, Bedford, MA, Tech. Rep. MTR-2547, 1973.

[8] K. J. Biba, Integrity Considerations for Secure Computer Systems MITRE Corporation, Bedford, MA, Tech. Rep. TR-3153, 1977.

[9] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE J. Selected Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[10] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles (SOSP'07)*, New York, 2007, pp. 321–334.

[11] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières, "Labels and event processes in the asbestos operating system," *ACM Trans. Comput. Syst.*, vol. 25, Dec. 2007.

[12] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. 11th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, New York, 2004, pp. 85–96, ACM.

[13] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proc. 34th Annu. Int. Symp. Computer Architecture (ISCA'07)*, New York, 2007, pp. 482–493, ACM.

[14] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. 12th Annu. Network and Distributed System Security Symp. (NDSS'05)*, 2005.

[15] F. Qin, C. Wang, Z. Li, H. S. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2006, pp. 135–148.

[16] D. J. Bernstein, Cache-Timing attacks on AES University of Illinois at Chicago, Chicago, IL, Tech. Rep. cd9faae9bd5308c440df50fc26a517b, 2005.

[17] O. A. Jean-Pierre, J. P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," in *Cryptology—CT-RSA 2007, Cryptographers Track at RSA Conf.*, 2007, pp. 225–242, Springer-Verlag.

[18] W.-M. Hu, "Reducing timing channels with fuzzy time," in *Proc. IEEE Computer Soc. Symp. Res. Security and Privacy 1991*, May 1991, pp. 8–20, IEEE.

[19] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn, "A retrospective on the VAX VMM security kernel," *IEEE Trans. Software Eng.*, vol. 17, no. 11, pp. 1147–1165, Nov. 1991.

[20] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proc. 14th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, New York, 2009, pp. 109–120.

[21] M. Tiwari, X. Li, H. Wassel, F. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture, MICRO-42. 2009*, New York, Dec. 2009, pp. 493–504.

[22] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I^2C and USB," in *Proc. 48th ACM/EDAC/IEEE Design Automation Conf. (DAC)*, San Diego, CA, Jun. 2011, pp. 254–259.

[23] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security," in *Proc. 38th Annu. Int. Symp. Computer Architecture (ISCA'11)*, New York, 2011, pp. 189–200, ACM.

[24] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *IEEE Trans. Computer-Aided Design of Integrated Circuits Syst.*, vol. 30, no. 8, pp. 1128–1140, Aug. 2011.

[25] R. Kastner, J. Oberg, W. Hu, and A. Irturk, "Enforcing information flow guarantees in reconfigurable systems with mix-trusted IP," in *Proc. Int. Conf. Eng. Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, NV, Jul. 2011.

[26] G. Micheli, "McGraw-Hill series in electrical and computer engineering: Electronics and VLSI circuits," in *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[27] B. Lin and S. Devadas, "Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, vol. 14, no. 8, pp. 974–985, Aug. 1995.

[28] F. Maamari and J. Rajski, "A method of fault simulation based on stem regions," *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.*, vol. 9, no. 2, pp. 212–220, Feb. 1990.

[29] M. R. Garey and D. S. Johnson*, Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1990.

[30] O. Ibarra and S. Sahni, "Polynomially complete fault detection problems," *IEEE Trans. Computers*, vol. C-24, no. 3, pp. 242–249, Mar. 1975.

[31] H. Fujiwara, "Computational complexity of controllability/observability problems for combinational circuits," *IEEE Trans. Computers*, vol. 39, no. 6, pp. 762–767, Jun. 1990.

[32] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Theoretical analysis of gate level information flow tracking," in *Proc. 47th ACM/IEEE Design Automation Conf. (DAC)*, Anaheim, CA, Jun. 2010, pp. 244–247.

[33] E. Dubrova, "Upper bound on the number of products in a sum-of-product expansion of multiple-valued functions," *Multiple-Valued Logic, Int. J.*, pp. 349–364, May 2000.

[34] E. J. McCluskey, "McGraw-Hill electrical and electronic engineering series," in *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill, 1965.

[35] E. B. Eichelberger, "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Development*, vol. 9, no. 2, pp. 90–99, Mar. 1965.

[36] L. Palopoli, F. Pirri, and C. Pizzuti, "Algorithms for selective enumeration of prime implicants," *Artificial Intell.*, vol. 111, no. 12, pp. 41–72, Jul. 1999.

[37] A. K. Chandra and G. Markowsky, "On the number of prime implicants," *Discrete Math.*, vol. 24, no. 1, pp. 7–11, 1978.

[38] T. Strzemecki, "Polynomial-time algorithms for generation of prime lmplicants," *J. Complexity*, vol. 8, pp. 37–63, 1992.

[39] W. V. Quine, "On cores and prime implicants of truth functions," *Amer. Mathematical Monthly*, vol. 66, pp. 755–760, 1959.

[40] N. N. Necula, "A numerical procedure for determination of the prime implicants of a boolean function," *IEEE Trans. Electron. Computers*, vol. EC-16, no. 5, pp. 687–689, Oct. 1967.

[41] M. Friedel, S. Nikolajewa, and T. Wilhelm, "The decomposition tree for analyses of boolean functions," *Math. Structures Computer Sci.*, vol. 18, pp. 411–426, 2008.

[42] E. Morreale, "Recursive operators for prime implicant and irredundant normal form determination," *IEEE Trans. Comput.*, vol. C-19, no. 6, pp. 504–509, Jun. 1970.

[43] B. Donald and O. Pederson, "Center for electronic systems design," Espresso: A multi-valued PLA minimization 1988 [Online]. Available: http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm

[44] S. H. Unger*, Asynchronous Sequential Switching Circuits*. Melbourne, FL: Krieger Publishing, 1983.

[45] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S. Nowick, "Synthesis of hazard-free customized cmos complex-gate networks under multiple-input changes," in *Proc. 33rd Design Automation Conf.*, June 1996, pp. 77–82.

[46] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, pp. 293–318, Sep. 1992.

[47] F. Somenzi, Cudd: Cu Decision Diagram Package 2011 [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/

[48] B. Donald and O. Pederson, "Center for electronic systems design," ABC: A System for Sequential Synthesis and Verification 2007 [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/

[49] "Berkeley logic synthesis and verification group," SIS: A System for Sequential Circuit Synthesis 2002 [Online]. Available: http://embedded.eecs.berkeley.edu/pubs/downloads/sis/index.htm

Wei Hu received the B.S. degree in pattern recognition and intelligent system from the School of Automation, Northwestern Polytechnical University, Xi'an, Shaanxi, China. He is currently pursuing the Ph.D. degree from the School of Automation, Northwestern Polytechnical University, Xi'an, Shaanxi, China.

His current research interests are in algorithm design and analysis, security, reconfigurable devices and embedded systems.

Jason Oberg (S'10) received the B.S. degree in computer engineering from the University of California, Santa Barbara. He is currently pursuing the Ph.D. degree, working with R. Kastner, from the Department of Computer Science and Engineering, University of California, San Diego.

His primary research interests include hardware and embedded system security with the use of information flow tracking.

Ali Irturk (S'07–M'10) received the Ph.D. degree in computer science and engineering from the University of California, San Diego.

He is currently a Research Scientist with the Department of Computer Science and Engineering, University of California, San Diego. His current research interests include design methods, languages and tools for embedded systems and their applications in areas, including signal processing, security, and high performance computing.

Mohit Tiwari received the Ph.D. degree in computer science from the University of California, Santa Barbara, in 2011.

He is currently a Computing Innovation Fellow at the University of California, Berkeley. His research interests include computer architecture and program analyses, especially applied to secure and reliable systems.

Dr. Tiwari's work has received the Best Paper Award at Parallel Architectures and Compilation Techniques in 2009 and the IEEE Micro Top Pick in 2010.

Timothy Sherwood (M'00) received the B.S. degree in computer science and engineering from the University of California (UC), Davis, in 1998, and the M.S. and Ph.D. degrees in computer science and engineering from UC, San Diego, in 2003.

He is currently an Associate Professor with the Department of Computer Science, UC, Santa Barbara. He specializes in the development of novel computer architectures for security, monitoring, and adaptive control.

Dr. Sherwood's papers have been selected as Micro Top Picks on four separate occasions, and he was the recipient of the 2009 Northrup Grumman Excellence in Teaching Award.

Dejun Mu received the Ph.D. degree in control theory and control engineering from Northwestern Polytechnical University, Xi'an, Shaanxi, China, in 1994.

He is currently a Professor with the School of Automation, Northwestern Polytechnical University, China. His current research interests include control theories and information security, including basic theories and technologies in network information security, application specific chips for information security, and network control systems.

Ryan Kastner (S'00–M'04) received the Ph.D. degree in computer science from the University of California, Los Angeles.

He is currently an Associate Professor with the Department of Computer Science and Engineering, University of California, San Diego. His current research interests include many aspects of embedded computing systems, including reconfigurable architectures, digital signal processing, and security.