Simultaneous Information Flow Security and Circuit Redundancy in Boolean Gates

Wei Hu*†, Jason Oberg†, Dejun Mu*, and Ryan Kastner†
*School of Automation, Northwestern Polytechnical University, Xi'an 710072, China
†Department of Computer Science and Engineering, University of California, San Diego
Email: {vinnie, mudejun}@nwpu.edu.cn; {jkoberg, kastner}@cs.ucsd.edu

***Abstract*—High assurance systems require strict guarantees on information flow security and fault tolerance or else face catastrophic consequences. Recently, Gate Level Information Flow Tracking (GLIFT) has been proposed to monitor information flows at the level of Boolean logic. At this level, all flows are explicit which makes it possible to detect security violations, even those that occur due to difficult to detect timing channels. In this paper, we show that the encoding technique used in previous GLIFT generation methods includes redundant encoding states, which leads to large overheads in area, delay and verification time. We present a new encoding technique with fewer encoding states by leveraging an inherent property of GLIFT. By denoting don't-care input conditions to logic synthesis tools, smaller GLIFT logic for dynamic information flow tracking is obtained and shorter simulation time for static information flow security verification is achieved. Experimental results using the *IWLS* benchmarks show average reductions of 39.8%, 31.1% and 57.5% in area, delay and simulation time respectively. Furthermore, the new encoding technique enables the GLIFT tracking logic to function both as information flow tracking and redundant logic. As a result, information flow security and fault tolerance can be simultaneously enforced with the same logic.**

I. Introduction

Information flow security lies at the heart of many security measures for high assurance systems involved in sensitive commercial or military information processing. Two common policies that usually need to be upheld in such systems are non-interference [1] and Bell LaPadula [2], which are frequently used to address data integrity and confidentiality. For integrity, it is required that an untrusted subsystem should never influence a trusted one, e.g., passengers should never be able to trigger an action in the flight control system from the user network on a commercial airline. For confidentiality, it demands that information never leaks from a classified subsystem to an unclassified one, e.g., a secret key for data encryption should never flow to an unclassified domain. A common technique to enforce data integrity and confidentiality is information flow tracking (IFT), which tracks information flows through the system to monitor whether or not any security policy is violated.

Previous IFT methods focus on tracking information flows at the program language (PL), operating system (OS), instruction set architecture (ISA) and microarchitecture (μARCH) levels. PL level IFT methods enforce information flow security through compile-time static verification using typing systems [3], [4]. Although these methods introduce little overhead in the final implementations, they force programmers to comply with new typing systems which lead to higher design complexity. OS level IFT methods monitor information flows with abstractions for operating system primitives such as processes, pipes and the file system [5]. They build information flow control mechanisms into the OS and thus can take the pressure of high design complexity off the programmers. However, these methods typically report up to 30% performance overhead (these coarse grained methods use byte or word level labels; the overhead will be even much higher if bit level labels are used) and, like PL level methods, they are at a too high level of abstraction to capture any hardware specific timing channels. ISA/μARCH level IFT implementations [6], [7] track information flows at the granularity of instruction and data words. They introduce very low performance overhead but are also at a level of abstraction that is transparent to timing behaviors in the underlying hardware. While information flows appear in various forms at PL, OS, ISA and μARCH levels, they can be precisely defined at the gate level in a way that unifies the notions of explicit flows, implicit flows, and even timing channels. Recently, to take a step forward in this direction, gate level information flow tracking (GLIFT) [8] has been proposed to understand how information flows from the level of Boolean gates.

Since GLIFT monitors information flows at the level of primitive Boolean operations, it is able to detect all logical information flows including those through timing channels. Timing channels in caches [9] and branch predictors [10] have previously been shown to leak secret encryption keys due to their nondeterministic latencies. There are *ad hoc* methods, such as clock fuzzing, to fix these very specific timing channels [11]; however, to the best of our knowledge, there has never been a systematic approach that can detect and ultimately eliminate them. GLIFT provides the first such methodology. These hardware specific flows can be detected and eliminated by taking a bottom-up approach to information flow security using GLIFT [12]. Although GLIFT provides an effective approach to enhance information flow security, fine grained IFT is inherently expensive either as a static verification technique or dynamic IFT approach. In the static verification scenario, significantly longer simulation time is required due to the huge size of the test vector space. In the dynamic scenario, previous work has observed up to 9X overhead in area and 3X overhead in delay [13]. While it is widely accepted that high design efforts and large area and delay overheads should be tolerated in high assurance systems, which demand strict enforcement of security properties, this paper aims to reduce such efforts and overheads *without* degrading security features by representing GLIFT logic in a more efficient manner.

Apart from information flow security, fault tolerance is another important factor that needs to be taken into account in design of high assurance systems. To guarantee the usability of such systems even when small portions of the system encounter a failure, techniques such as triple modular redundancy (TMR) [14] are often employed.

In this case, redundant logic is added in the physical design for fault tolerance, e.g., aircraft electronic flight controls utilize triple voting active redundancy [15]. This paper addresses both information flow security and fault tolerance by enforcing these two important properties with new GLIFT logic that is able to be configured for both IFT and circuit redundancy.

This paper proposes a new encoding scheme for GLIFT which leverages one of its special properties. This new scheme reduces the total number of encoding states and lead to smaller GLIFT logic for dynamic IFT and shorter simulation time for static information flow security verification. In addition to IFT, the new GLIFT logic can also be configured as a redundant circuit for fault tolerance. Specifically, this paper makes the following contributions:

- Leveraging an inherent property of GLIFT to derive a new encoding technique with reduced number of encoding states;
- Enabling simultaneous information flow tracking and circuit redundancy in Boolean logic;
- Presenting quantitative analysis using *IWLS* benchmarks to show reductions in terms of area, delay and verification time.

The remainder of this paper is organized as follows: Section II introduces the fundamentals of GLIFT, covering the basic concepts, the existing encoding technique for GLIFT and its drawbacks. In Section III, we propose an improved encoding technique, formalize GLIFT logic for Boolean gates using the new encoding, and discuss the improvements of the proposed encoding technique. Section IV presents experimental results in terms of area, delay and verification time using *IWLS* benchmarks. We conclude in Section V.

II. Fundamentals of GLIFT

A. How GLIFT Tracks Information Flows

GLIFT uses bit level labels to monitor the flow of information through the system. In GLIFT, each data bit in the *original logic* is associated with a label called *taint*, which indicates how this data bit should be tracked. Information about how the taint flows is maintained by the standalone *tracking or GLIFT logic*. Specifically, when the integrity of the data is considered, the taint bit will indicate if a data bit from the original logic is trusted/untrusted meaning that is either of high or low integrity; in the case of confidentiality, the taint bit will indicate if a data bit is secret/unclassified. Without loss of generality, a data bit is said to be *tainted* when its taint is logic '1' and *untainted* when its taint is logic '0'.

As an example, untrusted data can be labeled as tainted (trusted data marked as untainted) in integrity analysis. Taint propagates through GLIFT logic on the basis that the output of the tracking gate should be marked as tainted *iff* at least one of its tainted inputs has an influence at the output of the corresponding gate in the original logic. In this way, GLIFT more precisely captures actual information flows than previous conservative IFT methods which typically mark the output as tainted whenever there is *any* tainted input. We refer interested readers to [8], [13] for a more thorough discussion on the ideas behind GLIFT. In practice, there are two application scenarios of GLIFT, specifically *static information flow security verification* or *dynamic information flow tracking*.

In a static verification scenario [12], [16], [17], GLIFT is used to test or verify if a design contains any unintended flows that violate pre-defined information flow security policies. This is done at the system design stage. There is no need to physically instantiate the additional GLIFT logic after testing or verification completes, which prevents the area and delay overheads introduced by the standalone IFT logic. In this application scheme, the tester specifies what data to track, i.e., where tainted information initially arises. For example, the tester may set certain inputs to tainted and observes if security-critical portions of the design are affected by these tainted inputs; the tester can also set some secret data, e.g., private key, to tainted and observe if they leak to unclassified outputs. Depending on the specific security application, a large number of information flow testing or verification scenarios need to be run in order to guarantee at a high confidence that the design is secure.

In a dynamic IFT scenario [8], [18], GLIFT logic is physically instantiated during the design implementation phase to allow run-time checking of security properties. In this case, data are partitioned and associated with a label to indicate their secrecy or trustworthiness levels according to their origination. As an example, unprotected data coming from an open network environment are labeled as tainted to indicate that they are untrusted while data from a secure separation kernel will be marked as untained (trusted). Taints are then propagated through standalone GLIFT logic while data flow to outputs of the original design. These outputs will then be checked against their taints to determine if any tainted information flow to an output that is never expected to become tainted. The dynamic application scheme allows monitoring of more realistic execution patterns and real-time features that are hard to predict during design time. However, the area and delay overheads of GLIFT logic are significant due to the inherently high cost of fine grained IFT. Therefore, GLIFT logic used dynamically should be reserved for critical portions of the system such as data encryption units and routers/switches that multiplex between user and flight control networks on airlines. Dynamic GLIFT on these components would help mitigate the potential compromise without an impractical overhead on the system as a whole.

B. Existing Encoding Technique and Tracking Logic Generation Method for GLIFT

In the existing encoding technique [8], taint is independent from the logic value of the data bit, i.e., they are encoded separately. There are a total of four encoding states, namely *untainted '0'*, *untainted '1'*, *tainted '0'* and *tainted '1'*. For simplicity, we use symbols (U, 0), (U, 1), (T, 0) and (T, 1) to denote these states in the alphabet.

$$\alpha_1 = \{(U,0),(U,1),(T,0),(T,1)\}$$

The existing encoding technique assigns binary codes "00", "01", "10" and "11" to the symbols in α_1, respectively, and symbolic GLIFT logic for fundamental gates (AND/NAND, OR/NOR and NOT) are derived as a basis for constructively creating tracking logic [13]. The symbolic GLIFT logic functions are rewritten to logic equations as shown in (1) to (3). Here, $sh(f)$ is used to denote the GLIFT logic for function f and a_i is the taint of input A_i ($i = 1, 2, \cdots, n$); the product and sum operations are defined as logic AND and OR respectively and the $\sim$ symbol is the inverse operator.

$$sh(f = \prod_{i=1}^{n} A_i) = \prod_{i=1}^{n}(A_i + a_i)\cdot \sim \prod_{i=1}^{n} A_i\overline{a_i} \tag{1}$$

$$sh(f = \sum_{i=1}^{n} A_i) = \prod_{i=1}^{n}(\overline{A_i} + a_i)\cdot \sim \prod_{i=1}^{n} \overline{A_i}\overline{a_i} \tag{2}$$

$$sh(\overline{A_i}) = sh(A_i) = a_i \tag{3}$$

To generate GLIFT logic for circuits in polynomial time, one needs to build a library containing tracking logic for basic primitives, divide a given function into logic constructs and augment GLIFT logic for these subsections constructively in a manner similar to technology

mapping. In the existing GLIFT logic generation method, a library containing tracking logic for the AND/NAND, OR/NOR and NOT gates is constructed [13]. Such a library is functionally complete in generating GLIFT logic for any given Boolean circuit.

The existing encoding technique is capable of describing GLIFT logic. However, it contains extra encoding states resulting in area and delay overheads, which will be discussed in the next subsection.

C. Drawbacks of the Existing Encoding Technique

A major drawback of the existing encoding technique is that it contains redundant encoding states. This originates from an inherent property of GLIFT logic which states that the value of a tainted variable can be ignored in taint propagation. This property is an important premise to the successive discussions in this section.

It was shown that a data bit and its taint never appear in the same product term in simplified GLIFT logic [13]. In other words, the logic variable can be omitted from a product term that contains its taint. Assume the input A_i of function f is tainted, i.e., a_i = 1 (a_i is the taint of A_i) and use $sh(f)$ to denote the GLIFT logic of f. The input A_i can be omitted from $sh(f) \cdot a_i$ [13]. Thus, it can also be reduced from $sh(f)$ under the above assumption since $sh(f) = sh(f) \cdot a_i$ if $a_i = 1$ holds. When considering the GLIFT logic given in (1) to (3), if a_i is logic '1', A_i (either in its original or complement form) can be reduced from these equations. Because these equations are functionally complete for generating GLIFT logic for any Boolean circuit, the logic value of any tainted variable can be ignored in taint propagation. Thus, the tainted '1' and tainted '0' states in the existing encoding technique introduced in Section II-B can be combined to a single one to reduce the total number of encoding states to three. *Note that such state reduction is a consequence of a fundamental property of GLIFT logic derivation, and cannot be derived through state encoding and/or logic optimizations.*

From (1) to (3), one may notice that the number of product terms in the GLIFT logic for some logic primitives increases exponentially on the number of inputs in the worst case. As an example, according to (1), the number of product terms in the GLIFT logic for an n-input AND gate is $2^n - 1$. Consequently, large area and delay overheads are observed in the GLIFT logic represented in that manner [13]. In addition, such complex GLIFT logic also leads to longer simulation time for static information flow security verification.

Given that the existing GLIFT encoding technique has redundant encoding states, it tends to lead towards large overheads in area, delay, and verification time. We propose a more efficient encoding technique in the following section.

III. An Improved Encoding Technique for GLIFT

The improved encoding technique, originating from a fundamental property of GLIFT discussed in subsection II-C, completely changes the way in which GLIFT logic is described and designed. It provides a more efficient way for modeling taint propagation and enables the GLIFT circuit to function as both IFT and redundant logic, which can be used to make a system more secure *and* reliable.

A. An Improved Encoding Technique

As discussed in Section II-C, the logic value of a tainted variable can be ignored in taint propagation; the tainted '1' and tainted '0' states can be combined to a single one to reduce the total number of encoding states to three, namely *untainted '0'*, *untainted '1'* and *tainted*. We use symbols (U, 0), (U, 1) and (T, X) to denote these states in the new alphabet α_2.

$$\alpha_2 = \{(U,\ 0), (U,\ 1), (T,\ X)\}$$

Focusing on AND GLIFT logic in Table I, we can see that the intersection of the first two rows and first two columns correspond to untainted inputs. When both inputs are untainted, the outputs will be doubtlessly all untainted. And the data values are as expected, i.e., (U, 1) AND (U, 1) = (U, 1) while the other three combinations result in an untainted '0', i.e., (U, 0). The more interesting cases are shown in the third row and column which demonstrate the scenarios where at least one of the inputs is tainted. Here we can see that when one of the inputs is an untainted '0', i.e., (U, 0), the output is untainted, even when the other input is tainted. This is a consequence of the taint propagation rule for GLIFT, which enables GLIFT to more accurately capture actual information flow than previous conservative IFT methods. Finally, notice that whenever an output is marked as tainted (T), its data value is marked as a don't-care (X). The intuition here is that a tainted value must be assumed to be both '0' and '1', e.g., when tracking integrity, an untrusted (i.e., tainted) value could be untrustworthy and may switch from '0' to '1' or vice-versa. This intuition is the fundamental idea behind the reduction in symbols that we propose in this paper.

TABLE I
LOGIC AND OPERATION ON NEW ENCODING SYMBOLS.

AND	**(U, 0)**	**(U, 1)**	**(T, X)**
(U, 0)	(U, 0)	(U, 0)	(U, 0)
(U, 1)	(U, 0)	(U, 1)	(T, X)
(T, X)	(U, 0)	(T, X)	(T, X)

Consider GLIFT as a static verification technique. Under the old encoding technique, there are four encoding states. For the GLIFT logic of an n-input original Boolean function, there will be 4^n test vectors in the entire test vector space. In the new encoding technique, there are only three encoding states. Thus, the total number of test vectors can be reduced to 3^n, which is significant reduction in the size of the test vector space. In case a full coverage simulation is impractical, higher simulation coverage can be achieved when testing the same number of vectors under the new encoding technique.

Additionally, one can use three-valued simulation technique to accelerate static verification under the new encoding. This can be achieved by assigning values of '0', '1' and 'X' to the symbols (U, 0), (U, 1) and (T, X) respectively, where 'X' represents the frequently used nondeterministic state in circuit simulation. It can be verified that the logic operation rules defined on these values agree with the taint propagation rules defined on the symbols. Further, three-valued logic simulation can be performed directly on the original logic; there is no need to generate additional GLIFT logic for static verification. Since GLIFT logic typically dominates the original circuit in complexity, e.g., a larger number of gates primitives, three-valued simulation on the original circuit can be more efficient. The new encoding technique, which reduces the total number of encoding state to three, reduces the size of the test vector space and enables three-valued simulation on the original circuit directly, can significantly speed-up static information flow security verification using GLIFT.

When GLIFT logic is physically implemented for dynamic IFT, at least two Boolean bits are needed to encode three states. This leads a *don't-care* input set since two Boolean bits can encode a maximum number of four states. Such *don't-care* input conditions will not affect the functionality of the GLIFT logic or lead to a security policy violation because they never appear at the inputs. However, denoting these *don't-care* input conditions to the logic synthesis tools will result in better implementation results. As a sanity check, we formalize GLIFT logic for the two-input AND gate

without and *with* considering the *don't-care* input set respectively, when symbols (U, 0), (U, 1) and (T, X) are encoded as "00", "01" and "10". These are shown in (4) and (5), where $a_{[1:0]}$, $b_{[1:0]}$ and $o_{[1:0]}$ denote the encoding results, namely labels of A, B and O, e.g., $a_{[1:0]}$ will indicate if input A is (U, 0), (U, 1) or (T, X).

$$\begin{aligned} o_1 &= a_1\overline{a_0}b_1\overline{b_0} + \overline{a_1}a_0b_1\overline{b_0} + a_1\overline{a_0}\overline{b_1}b_0; \\ o_0 &= \overline{a_1}a_0\overline{b_1}b_0; \end{aligned} \tag{4}$$

$$\begin{aligned} o_1 &= a_1b_1 + a_0b_1 + a_1b_0 \\ o_0 &= a_0b_0 \end{aligned} \tag{5}$$

From (4) and (5), denoting *don't-care* input conditions to logic synthesis tools will result in significantly smaller GLIFT logic. Such conclusion also holds when the symbols are assigned to other binary codes. By comparison, the old encoding inherently has four encoding states and will not benefit from such *don't-care* inputs. Thus, the new encoding reduces the total number of encoding states and further leads to optimized GLIFT logic.

B. GLIFT Logic for Boolean Gates under the New Encoding

The new encoding uses two Boolean bits to encode three states. One needs to select three binary codes out of four for the encoding symbols. Consequently, there are a total of 24 possible encoding schemes. It is impossible to find an encoding scheme that is optimal for all GLIFT circuits because the problem is hard in nature [19] and optimal encodings are usually specific to given circuits. However, it is possible to perform area and delay analysis on GLIFT logic for a set of basic Boolean operators under different encoding schemes and choose a relatively better one. After testing all 24 possible encoding schemes *with* consideration of the *don't-care* input set, those that report the smallest area and delay are shown in Table II. As an example, in the first encoding scheme, (U, 0), (U, 1) and (T, X) are encoded to be "00", "11" and "01" respectively.

TABLE II
ENCODINGS THAT REPORT THE SMALLEST AREA AND DELAY.

Encodings	**(U, 0)**	**(U, 1)**	**(T, X)**
Enc. 1	00	11	01
Enc. 2	00	11	10
Enc. 3	11	00	01
Enc. 4	11	00	10

Test results show that the four different encoding schemes in Table II result in exactly the same GLIFT logic for primitive gates. Thus, we choose *Enc. 1* for further analysis. Subsequently, the GLIFT logic for the AND, OR and NOT gates can be derived. Consider n-input gates with inputs $A1, A2 \cdots An$, the GLIFT logic for the AND and OR gates can be formalized as shown in (6) and (7) respectively, where $a^i_{[1:0]}$ denotes the label of Ai.

$$\begin{aligned} o_1 &= a_1^1 \cdot a_1^2 \cdots a_1^n \\ o_0 &= a_0^1 \cdot a_0^2 \cdots a_0^n \end{aligned} \tag{6}$$

$$\begin{aligned} o_1 &= a_1^1 + a_1^2 + \cdots + a_1^n \\ o_0 &= a_0^1 + a_0^2 + \cdots + a_0^n \end{aligned} \tag{7}$$

To maintain a functionally complete GLIFT library for constructing tracking logic for digital circuits, at least the GLIFT logic for the NOT gate should be included. This is shown in (8).

$$\begin{aligned} o_1 &= \overline{a_0} \\ o_0 &= \overline{a_1} \end{aligned} \tag{8}$$

It is important to notice that o_1 gets the inverse of a_0 and o_0 gets the inverse of a_1 in (8), which adheres to our encoding scheme, namely the inverse of (T, X) remains as "01".

The GLIFT logic for n-input NAND and NOR gates can be formalized as that for n-input AND and OR gates followed by the GLIFT logic for the NOT gate. These are given in (9) and (10) respectively.

$$\begin{aligned} o_1 &= \overline{a_0^1 \cdot a_0^2 \cdots a_0^n} \\ o_0 &= \overline{a_1^1 \cdot a_1^2 \cdots a_1^n} \end{aligned} \tag{9}$$

$$\begin{aligned} o_1 &= \overline{a_0^1 + a_0^2 + \cdots + a_0^n} \\ o_0 &= \overline{a_1^1 + a_1^2 + \cdots + a_1^n} \end{aligned} \tag{10}$$

From (6) to (10), the GLIFT logic for n-input AND/NAND and OR/NOR gates is two product terms whose size is linear to n. This is different from the old encoding technique, where the number of product terms increases exponentially to n as shown in (1) and (2). Additionally, the GLIFT logic represented in the new encoding technique has a constant one logic level even for an n-input gate. By comparison, logic levels in the GLIFT logic for AND/NAND and OR/NOR represented using the old encoding technique increases linearly to n.

Figure 1 (a) to (c) and (d) through (h) show the GLIFT logic for primitive gates represented in the old and new encoding techniques respectively. As shown in Fig. 1 (c) and (h), the old encoding technique simply uses a wire while the new encoding uses two gates to represent the GLIFT logic for the NOT gate. Therefore, new GLIFT logic tends to report larger area and delay if the original design consists of more NOT gates than AND/NAND and OR/NOR gates. From Fig. 1 (d) to (g), the new GLIFT logic for the AND/NAND and OR/NOR gates only introduces an additional gate without increasing logic levels. There can result in significant reductions in area and delay as compared to the corresponding old GLIFT logic in Fig. 1 (a) and (b).

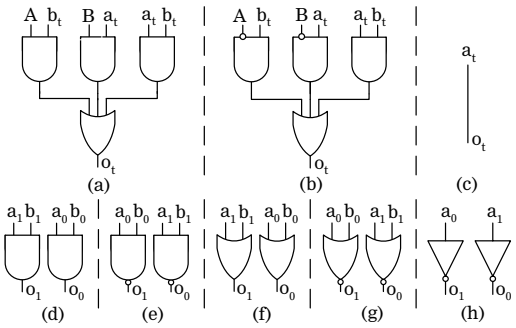

Fig. 1. (a) to (c): GLIFT logic for AND/NAND, OR/NOR and NOT gate using the old encoding; a_t, b_t and o_t are the taints of A, B and O respectively. (d) to (h): GLIFT logic for AND, NAND, OR, NOR, and NOT gate using the new encoding; $a_{[1:0]}$, $b_{[1:0]}$ and $o_{[1:0]}$ are the input and output labels.

Besides the AND, OR, NAND and NOR gates, the new GLIFT logic for more complex Boolean functions can also be far less complicated. For a concrete understanding, consider the two-input multiplexer (MUX-2), whose logic function is $O = SA + \overline{S}B$. To generate GLIFT logic for MUX-2 using the old encoding, one needs to divide the function into two AND gates and an OR gate and augment tracking logic for them constructively. The simplified resulting circuit is given in (11).

$$o_t = Sa_t + \overline{S}b_t + A\overline{B}s_t + \overline{A}Bs_t + a_ts_t + b_ts_t \tag{11}$$

Under the new encoding, this process is more straightforward. One can directly write out the new GLIFT logic as shown in (12).

$$\begin{aligned} o_1 &= s_1 a_1 + \overline{s_0} b_1 \\ o_0 &= s_0 a_0 + \overline{s_1} b_0 \end{aligned} \tag{12}$$

By the comparison, GLIFT logic for n-input AND, OR, NAND, NOR and other function units can be much simpler. As a result, GLIFT logic represented in the new encoding technique tends to report significantly smaller area and delay, which we will show in the results section. In addition, the new GLIFT logic can also be configured as a redundant circuit of the original design for fault tolerance. This is covered in the next subsection.

C. Enabling Circuit Redundancy

Another benefit of the new encoding is that it enables GLIFT logic to function as a redundant circuit. This originates from the observation that *when no input is tainted, the new GLIFT logic will behave exactly the same as the original design.*

Under the new encoding scheme, inputs to the GLIFT circuit will take the values of the original variables when they are untainted, e.g., both a_1 and a_0 will take the value of A. Further, if no input is tainted, only (U, 0) and (U, 1) will be propagating in the GLIFT logic; (T, X) will not appear. By incrementally denoting (T, X) as *don't-care* to logic synthesis tools, (8) can be rewritten as (13). In this case, the new GLIFT logic will be *functionally but not physically* exactly twice the original circuit. In other words, when no input is tainted, the new GLIFT logic will function as a redundant circuit.

$$\begin{aligned} o_1 &= \overline{a_1} \\ o_0 &= \overline{a_0} \end{aligned} \tag{13}$$

For a more concrete understanding, consider the original circuit and new GLIFT logic for MUX-2 as shown in Fig. 2. If no input is tainted, we have $a_1 = a_0 = A$, $b_1 = b_0 = B$ and $s_1 = s_0 = S$. When these values are propagated through the GLIFT logic, the same value will be observed at the outputs of both the original circuit and GLIFT logic, i.e., $o_1 = o_0 = O$. In other words, GLIFT logic operates as a two copies of the original logic function.

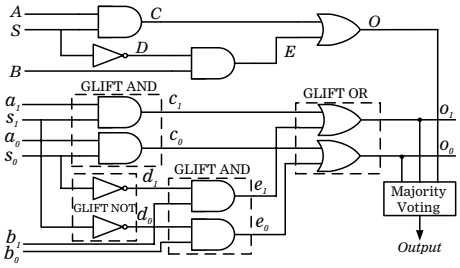

Fig. 2. The original circuit and GLIFT logic for MUX-2. The outputs o_1 and o_0 will be equal to O when no input is tainted. In this case, the new GLIFT logic functions as circuit redundancy.

This highlight of the new encoding technique can be quite useful because high assurance systems often require circuit redundancy for fault detection and tolerance. To detect a fault, one sets all the inputs to untainted, i.e., (U, 0) or (U, 1) and checks if the original circuit and GLIFT logic have identical outputs. If a fault is detected in the original circuit, the GLIFT logic can be temporarily used as substitute. Further, the TMR [14] can be implemented by adding a majority voter at the output stage since there are two redundant bits for each output as shown in Fig. 2.

IV. Experimental Results

We carried out experiments on several *IWLS* benchmarks [20] to obtain area, delay and simulation time results.

For simulation time, GLIFT logic in the old encoding is generated using our own script; the original circuit is used directly as the GLIFT logic under the new encoding and three-valued simulation technique is employed for static verification. Both designs are simulated using *ModelSim* to a total number of 2^{24} random vectors generated by LFSR (Linear Feedback Shift Register). For each execution, a $2n$-bit LFSR and an n-bit LFSR are used to generate the random vectors for the GLIFT logic in the old and new encodings respectively. The nondeterministic 'X' state is inserted into the test vectors for the new GLIFT circuit using additional glue logic. In our test, different initial seeds are denoted to the LFSRs to verify that the results are consistent and independent from the initials.

Figure 3 shows the simulation time results in seconds (sec). *Reductions* in simulation time are given in percentage. As an example, the new encoding technique reduces simulation time by 67.1% for the benchmark *alu2*. On average, 57.5% reductions in simulation time are achieved on all the benchmarks tested. The results show the complexity of the benchmarks as a whole (number of I/O's, gates, and logic levels).

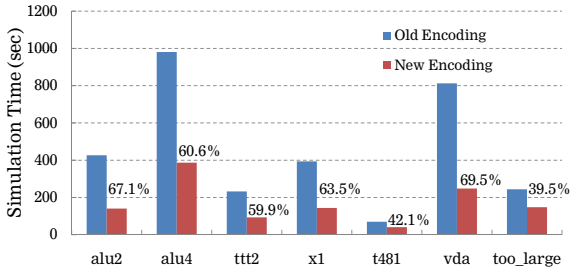

Fig. 3. Simulation time of GLIFT logic circuits in both encodings. The percentage data show reductions in simulation time.

We then represent GLIFT logic in both encodings. The designs are synthesized using *Synopsys Design Compiler* and targeted to its *90nm* standard cell library [21] for area and delay reports, as shown in Table III. The area results are converted to the number of NAND gates. *Reductions* in area and delay are given in percentage.

From Table III, the GLIFT logic represented in the new encoding technique reports significantly smaller area and delay for most benchmarks, especially the larger ones. As an example, the GLIFT logic for *alu2* described using the old and new encoding techniques has area/delay of 1326/1.88 and 717/1.26 respectively; this is 45.9% reduction in area and 33.0% reduction in delay. On average, the GLIFT logic represented in the new encoding reduces the area by 39.8% and delay by 31.1%. The improvement in area is a consequence of the simpler GLIFT logic for basic logic gates, especially for large AND, OR, NAND and NOR gates. In addition, the new encoding technique also reduces the number of logic levels and thus achieves significantly smaller delay results. For *t481*, its GLIFT logic in the new encoding reports larger area. This is because this benchmark contains over 30% inverters (when synthesized for the target GLIFT library) and the new GLIFT logic for the inverter is more complicated according to Fig. 1 (c) and (h).

Due to the inherently high cost of fine grained IFT, GLIFT logic will typically dominate the original Boolean circuit in area and delay. Although these overheads are reduced to roughly twice the original design using our novel encoding technique, they can be

TABLE III
AREA AND DELAY RESULTS OF BOTH THE ORIGINAL CIRCUIT AND GLIFT LOGIC CIRCUITS. AREA IS CONVERTED TO THE NUMBER OF NAND GATES; DELAY RESULTS ARE IN NANOSECONDS (*ns*).

Benchmark	# of I/O	% NOT	Area	Delay	Area			Delay		
					OldEnc.	NewEnc.	Reduc.	OldEnc.	NewEnc.	Reduc.
ttt2	24/21	14.4%	146	0.32	498	388	22.1%	0.53	0.52	1.90%
alu2	10/6	6.16%	312	0.97	1326	717	45.9%	1.88	1.26	33.0%
alu4	14/8	10.9%	591	1.30	2686	1261	53.1%	2.22	1.35	39.2%
vda	17/39	3.99%	665	0.58	2955	1551	47.5%	1.24	0.87	29.8%
x1	51/35	12.0%	233	0.31	1492	646	56.7%	0.76	0.40	47.4%
t481	16/1	32.6%	44	0.22	68	76	-11.8%	0.26	0.21	19.2%
too_large	38/3	12.9%	246	0.48	1537	627	59.2%	1.34	0.61	54.5%
pair	173/137	10.7%	1600	0.66	8298	4574	44.9%	1.45	0.89	38.6%
i10	257/224	14.4%	2050	1.93	8205	5505	32.9%	4.47	2.33	47.9%
C3540	50/22	10.8%	1336	1.50	5288	2868	45.8%	2.46	1.95	20.7%
C5315	178/123	13.1%	1808	1.10	7653	5239	31.5%	1.66	1.48	10.8%
C6288	32/32	5.85%	7526	5.47	58038	18016	69.0%	12.1	6.59	45.5%
C7552	207/108	16.4%	1089	1.21	6603	5723	13.3%	4.05	2.59	36.0%
DES	256/245	5.32%	3358	0.79	14981	7917	47.2%	1.42	1.26	11.3%
Average Reductions							39.8%			31.1%

still expensive for hardware fabrication. In practice, there are usually partitions of security domains across the design and GLIFT logic only needs to be deployed for the security critical portions, e.g., there is no need to track information flows within the user network on a commercial airline. This will significantly decrease the total area and delay overheads. Further, at least twice hardware redundancy is required for standard fault detection and tolerance (e.g., TMR). The new encoding technique enables both IFT and circuit redundancy through shared hardware resource, which prevents additional area and delay overheads introduced by instantiating these as separate logic.

From the experimental results, the new encoding technique is far more efficient in describing taint propagation because of the significant reductions in area, delay and simulation time for most benchmarks. Such reductions are achieved through a complete change to the alphabet, encoding scheme and the way in which GLIFT logic is designed and verified. This new encoding technique could not be derived by synthesis tools using state encoding and/or logic optimizations. These improvements are essential when GLIFT is to be used to statically verify or dynamically track information flows in high assurance systems.

V. CONCLUSION

GLIFT is able to account for all logical information flows including those through timing channels. It provides an effective approach to monitor and enforce important security properties of a system. This paper proposes a novel encoding technique for GLIFT and significantly reduces the overheads in area, delay and verification time. In addition, the new encoding enables the GLIFT logic to be configured as redundant circuit for fault tolerance, which further prevents area and delay overheads since no additional logic needs to be deployed as circuit redundancy. The new encoding described herein is not merely an assignment of new binary codes to different encoding states under the previous encoding technique. Conversely, we leverage an inherent property of GLIFT that the value of a tainted variable can be ignored in taint propagation and bring forward a fundamental change in the way that the GLIFT logic is described.

REFERENCES

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," *IEEE Symp. Security and Privacy*, April 1982, pp. 11-20.

[2] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," Tech. Rep. MTR-2547, MITRE Corp., Bedford, MA, 1973.

[3] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Computer Security*, 4(2-3), 1996, pp. 167-187.

[4] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Trans. on Programming Languages and Systems*, 25(1), Jan. 2003, pp. 117-158.

[5] M. Krohn, A. Yip, M. Brodsky, N. Cliffer and M. F. Kaashoek et al., "Information Flow Control for Standard OS Abstractions," In *Proc. of ACM SIGOPS Operating Syst. Rev.*, Vol. 41, October 2007, pp. 321-334.

[6] G. E. Suh, J. W. Lee, D. Zhang and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," In *Proc. 11th ASPLOS*, 2004, pp. 85-96.

[7] M. Dalton, H. Kannan and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," In *Proc. 34th ISCA*, Jun. 2007, pp. 482-493.

[8] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong and T. Sherwood, "Complete information flow tracking from the gates up," In *Proc. 14th ASPLOS*, 2009, pp. 109-120.

[9] D. J. Bernstein, "Cache-timing attacks on AES," Technical Report, 2005.

[10] O. Accigmez, J. pierre Seifert and C. K. Koc, "Predicting Secret Keys via Branch Prediction," In *Cryptology, The Cryptographers Track at RSA*, Springer-Verlag, 2007, pp. 225-242.

[11] W. M. Hu, "Reducing Timing Channels by Fuzzy Time," In *Proc. Symp. on Res. Security Privacy*, 1991, pp. 8-20.

[12] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, "Information Flow Isolation in I2C and USB," in *Proc. DAC*, 2011, pp. 254-259.

[13] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, "Theoretical Fundamentals of Gate Level Information Flow Tracking," *IEEE Trans. on CAD*, VOL. 30(8), August 2011, pp. 1128-1140.

[14] D. S. Herrmann, *A Practical Guide to Security Engineering and Information Assurance,* Boca Raton: CRC Press, Auerbach, 2001.

[15] P. D. T. O'Connor, D. Newton and R. Bromley, *Practical reliability engineering,* 4th Edition, Wiley, Chichester, 2002.

[16] M. Tiwari, J. Oberg, X. Li, J. K. Valamehr and T. Levin et al., "Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security", in *Proc. 38th ISCA)*, June 2011, pp. 189-200.

[17] R. Kastner, J. Oberg, W. Hu and A. Irturk, "Enforcing Information Flow Guarantees in Reconfigurable Systems with Mix-trusted IP," in *Proc. ERSA*, 2011.

[18] M. Tiwari, X. Li, H. M. G. Wassel, F. T. Chong and T. Sherwood, "Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference," In *Proc. Int. Symp. Micro*, 2009, pp. 493-504.

[19] K.J. Adams, J.G. Campbell, L.P. Maguire and J.A.C. Webb, "State assignment techniques in multiple-valued logic," in *Proc. 29th IEEE Int. Symp. Multiple-Valued Logic*, May 1999, pp. 220-225.

[20] "IWLS 2002 Benchmarks," Ver 1.0, [Online]. Available: http://iwls.org/iwls2002/benchmarks.html

[21] Synopsys, "SAED_EDK90_CORE - 90nm Digital Standard Cell Library," Reversion. 1.8, [Online]. Available: http://www.synopsys.com/community/universityprogram/pages/library.aspx