An Improved Encoding Technique for Gate Level Information Flow Tracking

Wei Hu[†,‡], Jason Oberg[‡], Ali Irturk[‡], Mohit Tiwari[§], Timothy Sherwood[§], Dejun Mu[†], and Ryan Kastner[‡]
[†]Automation, Northwestern Polytechnical University, Xi'an, China
[‡]Computer Science and Engineering, University of California, San Diego
[§]Computer Science, University of California, Santa Barbara
{w3hu, jkoberg, airturk, kastner}@cs.ucsd.edu, {tiwari, sherwood}@cs.ucsb.edu, mudejun@nwpu.edu.cn

Abstract—**High-assurance systems, such as flight control and banking systems require strict guarantees on information flows or else face catastrophic consequences. Information flow tracking (IFT) is a frequently used security measure for preventing unintended information flows in such systems. Recently, Gate Level Information Flow Tracking (GLIFT) has been proposed to track information flows at the hardware level. GLIFT enables a concrete understanding of all information flows from Boolean gates. It unifies the notions of explicit flows, covert channels, and even timing channels at the gate level and provides a general approach for enhancing important security properties such as integrity and confidentiality. This article presents a new encoding scheme for GLIFT with fewer encoding states by combining two states into one. Unlike the previous method, this reduction in encoding states allows the GLIFT tracking logic to operate independently from the original circuit. This independence allows for the GLIFT logic to be configured as both redundancy and tracking logic for the original circuit. Further, experimental results show this new state assignment provides on average 25.7% reductions in area, 31.4% reductions in delay, and 48.6% decrease in simulation time for several IWLS benchmarks.**

I. Introduction

High assurance systems used for sensitive financial or military information processing all demand a level of security far beyond the norm. Two common security 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. The non-interference policy requires that an untrusted sub-system 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 an airplane. The Bell LaPadula policy demands that information never leaks from a classified sub-system to an unclassified one, e.g., a secret key for data encryption should never flow to portions other than cipher text. While these are very strong and useful policies, they are difficult to deploy and even harder to verify in practice. Among the different approaches to enforce integrity and confidentiality, information flow tracking (IFT) is a frequently used technique due to its efficiency in detecting unintended information flows.

IFT provides an effective approach for preventing unintended interaction between different components in a system. However, previous IFT methods tend to force programmers to comply with new typing systems and design rules that lead to higher design complexity [3], [4], introduce large overheads to system performance [5] or use coarse granularity labels and propagation policies which are overly conservative [6], [7]. In addition, they all ignore hardware specific side channels. While information flows appear in various forms at program language (PL), operating system (OS) and instruction set architecture (ISA) levels, they can be precisely defined in a way that unifies the notions of explicit flows, covert channels, and even timing channels at the gate level.

Gate level information flow tracking (GLIFT) [8] provides a concrete understanding of how information flows through AND, OR, NOT and other Boolean gates, and all the way up to the system stack. It is able to detect all logical flows including those through hardware specific side channals such as timing channels. Timing channels in caches [9] and branch predictors [10] have previously been shown to leak secret keys due to their nondeterministic latencies. There are ad hoc methods to fix these very specific timing channels such as clock fuzzing [11], but there has never been a systematic approach that can detect and ultimately eliminate them. GLIFT provides the first such methodology. By taking a bottom-up approach to information flow security using GLIFT, these hardware specific flows can be eliminated. However, we have observed that GLIFT logic described using the current method requires intermediate wires from the original circuit, causing the GLIFT logic and original circuit to be nested. This would increase the complexity for circuit design and verification. Additionally, the previous GLIFT logic generation method encodes a data bit and its label separately [8], which leads to extra number of encoding states and further, area and delay overheads.

This article proposes a new encoding scheme for GLIFT and reduces the total number of encoding states. Consequently, GLIFT logic described using the new encoding is self-contained and independent from the original circuit. Apart from information flow tracking, the new GLIFT logic can function as circuit redundancy for fault tolerance. Experimental results using IWLS benchmarks have also shown significant reduction in area, delay and simulation time. Specifically, this article makes the following contributions:

- Proposing a new state assignment for GLIFT with reduced number of encoding states;
- Enabling both information flow tracking and circuit re-

dundancy without introducing extra logic;

- Presenting quantitative analysis of the GLIFT logic for *IWLS* benchmarks to show reductions in terms of area, delay and simulation time.

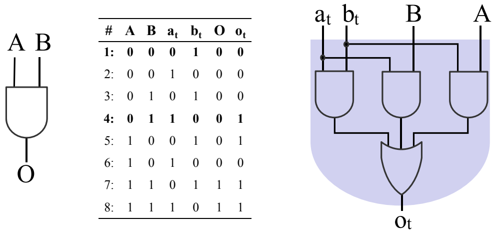
The remainder of this article is organized as follows: Section II introduces the fundamentals of GLIFT, covering the basic concepts, the existing encoding technique for GLIFT logic representation and its drawbacks. In Section III, we propose an improved encoding technique for GLIFT, and perform a comparison to the existing scheme. Section IV presents experimental results in terms of area, delay and simulation time using *IWLS* benchmarks. We conclude in Section V.

II. Fundamentals of GLIFT

This section introduces how data bits are labeled in GLIFT and illustrates how information flows are tracked at the gate level with a simple example. It also discusses how GLIFT logic is currently defined and constructed. Finally, the drawbacks of the existing encoding technique are covered.

A. How GLIFT Tracks Information Flows

In information flow analysis, data are usually associated with a label indicating their trustworthiness or security level. This label is propagated through the system under pre-defined policies and checked to prevent unintended information flows. GLIFT uses fine granularity labels and propagation policies. Each data bit is associated with a tag called *taint*. A logic variable is said to be *tainted* when its taint is logic true and *untainted* when its taint is logic false. Taint is propagated from the input to the output of a function if the tainted input has an influence on the output. As an example, consider the two-input AND gate (AND-2) in Figure 1 (a).



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

(a) (b) (c)

Fig. 1. (a) A two-input AND gate. (b) Partial truth table of AND-2 with taint information. (c) The GLIFT logic of AND-2 is $Ab_t + Ba_t + a_tb_t$.

Figure 1 (b) shows the partial truth table of AND-2 with taint information, where a_t, b_t and o_t are the taints of A, B and O respectively. When both inputs of AND-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, let's consider the first row ($A = 0, B = 0, a_t = 0, b_t = 1$). When changing the value of the tainted input B, the output does not change. Thus, the tainted input does not have an influence on the output and the output should be marked as untainted ($o_t = 0$). In this case, the untainted '0' which dominates the logic value at the output and prevents tainted information from flowing to the output. Then let's consider row 4 ($A = 0, B = 1, a_t = 1, b_t = 0$). When changing the value of the tainted input A, a change at the output will be observed. Thus, the tainted input has an influence on the output and the output should be marked as tainted ($o_t = 1$). Once the GLIFT logic for AND-2 is simplified, the resulting circuit is shown in Figure 1 (c).

Upon the basic concepts, the next subsection introduces the currently used encoding technique and tracking logic generation method for GLIFT.

B. Existing Encoding Technique and GLIFT Logic Generation Method

In the existing encoding technique [8], taint is independent from the logic value of the data bit since they are encoded as separate bits. Using this encoding technique, previous work [12] has formalized GLIFT logic for AND, OR and NOT gates as shown in Equation 1 to Equation 3 in order to create GLIFT logic for circuits constructively with these building blocks. Here, $sh(f)$ is used to denote the GLIFT logic for logic function f and a_i is the taint of A_i ($i = 1, 2, \cdots, n$). The minus sign in the equations means removing the last term from the expression.

$$sh(\overline{f}) = sh(f) \tag{1}$$

$$sh(f = A_1 \cdot A_2 \cdots A_n) = \prod_{i=1}^{n}(A_i + a_i) - f \tag{2}$$

$$sh(f = A_1 + A_2 + \cdots + A_n) = \prod_{i=1}^{n}(\overline{A_i} + a_i) - \overline{f} \tag{3}$$

To generate GLIFT logic for a circuit in linear time, one needs to build a library containing tracking logic for basic primitives, divide a given function into logic constructs and generate GLIFT logic for these subsections constructively in a manner similar to technology mapping. In the existing GLIFT logic generation method [8], a library containing tracking logic for the AND, OR and NOT gates is constructed. 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 and further, area and delay overheads, which will be discussed in detail in the next subsection.

C. Drawbacks of the Existing Encoding Technique

In the existing encoding technique, a data bit can be either '0' or '1' while its taint can be either *tainted* or *untainted*. Thus, there are four encoding states in the existing technique. However, as proved by [12], a data bit and its taint never appear in the same product term in simplified GLIFT logic. In other words, the logic variable can be omitted from a product term that contains its taint. Let's assume the input A of function f is tainted, i.e., $a_t = 1$ and use $sh(f)$ to denote the GLIFT logic of f. We have $sh(f) = sh(f) \cdot a_t$, since $a_t = 1$. As a result, the input A can be omitted from $sh(f) \cdot a_t$. In other words, the logic value of a tainted variable can be ignored in taint propagation. Thus, we can combine the tainted '1' and tainted '0' states in the existing encoding technique to a single one for encoding state reduction.

Additionally, from the encoding technique and GLIFT logic generation method introduced, one may notice that the number of product terms in the GLIFT logic for some logic primitives increases exponentially to the number of inputs. As an example, according to Equation 2, the number of product terms in the GLIFT logic for an n-input AND gate is $2^n - 1$. Such an exponential increase in the number of product terms leads to complex GLIFT logic for basic gates and further even more complex GLIFT logic for circuits. As a consequence, large area and delay overheads are observed in the GLIFT logic represented in that manner [12]. Further, the exponential increase of product terms in GLIFT logic represented also leads to long simulation time for design verification.

As mentioned, data bits and their taints are encoded separately in the existing encoding scheme. As a result, both the data bits and their taints need to be present in the GLIFT logic. More specifically, GLIFT logic needs to reference intermediate wires from the original design, which leads to a nested design of the original circuit and its GLIFT logic. This causes the resulting circuit to be harder to optimize and verify.

Since the existing encoding technique has extra number of encoding states, large overheads in area, delay, and high design complexity, we propose a more efficient encoding technique in the following section.

III. An Improved Encoding Technique for GLIFT

The improved encoding technique reduces the total number of encoding states. In addition, it separates the GLIFT logic from the original design and enables the GLIFT logic circuit to function as both tracking logic and circuit redundancy.

A. An Improved Encoding Technique

As described 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*. For simplicity, we use symbols (U, 0), (U, 1) and (T, X) to denote these states respectively.

In the binary implementation of GLIFT logic, at least two Boolean bits are needed to encode three states. Consequently, there are a total of 24 possible encoding schemes. It is impossible to find an encoding technique that is optimal for all circuits because the problem is hard in nature [13] and optimal encodings are usually specific to given circuits. However, since GLIFT logic is constructed using tracking logic for Boolean gates, it is possible to perform area and delay analysis on GLIFT logic for the basic constructs under different encoding schemes. After testing all 24 possible encoding schemes, those that report the smallest area with a short delay (those that have a slightly shorter delay report significantly larger area) are shown in Table I. As an example, in the first encoding scheme, (U, 0), (U, 1) and (T, X) are encoded to be "00", "11" and "01".

TABLE I
New Encodings with the smallest area and a balanced delay.

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

To distinguish from the old encoding technique, the new one uses a different way to denote encoding results. For a given variable, we use its name with subscripts of 1 and 0 to denote the two-bit encoding result for that variable. As an example, the two-bit encoding result of a given variable A is denoted by A_1 and A_0. In the next subsection, we will create GLIFT logic for the AND, OR and NOT gates using the new encoding technique.

B. GLIFT Logic for Boolean Gates under the New Encoding

Test results have shown that the four different encoding schemes in Table I share exactly the same GLIFT logic for the AND, OR and NOT gates. Thus, we choose the first encoding scheme for further analysis. Table II and Table III show the results of logic AND and OR operations on the new symbols. As an example, the AND operation results of (U, 0) with (U, 0), (U, 1) and (T, X) are all (U, 0) as shown by row 2 in Table II.

TABLE II
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)

TABLE III
Logic OR operation on new encoding symbols.

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

From these two tables we can discover that the new symbols are compatible with the rules defined for the logic AND and OR operations. Thus, the GLIFT logic for the AND and OR gates will be simply two AND and OR gates respectively. However, AND gate together with OR gate do not make up a complete function set that is able to describe all logic circuits. To make it complete, at least the NOT gate should be included. Unfortunately, the new symbols are incompatible with the rules defined for logic NOT in that the inverse of "01" becomes "10", which is not defined. Thus, we need to study the logic NOT operation on the new encoding symbols as defined in Table IV and formalize GLIFT logic for the NOT gate.

TABLE IV
LOGIC NOT OPERATION ON NEW ENCODING SYMBOLS.

NOT	**(U, 0)**	**(U, 1)**	**(T, X)**
	(U, 1)	(U, 0)	(T, X)

Let's denote the two inputs to GLIFT logic for a NOT gate by $A_{[1:0]}$ and the outputs by $O_{[1:0]}$. From Table IV, one can formalize the following GLIFT logic for the NOT gate.

$$\begin{aligned} O_1 &= \overline{A_0} \\ O_0 &= \overline{A_1} \end{aligned} \tag{4}$$

It is important to notice that O_1 gets the inverse of A_0 and O_0 gets the inverse of A_1 which adheres to our encoding technique, namely the inverse of (T, X) remains as "01".

When considering n-input gates with inputs $A1, A2 \cdots An$, the GLIFT logic for AND and OR gates can be formalized as shown in Equation 5 and Equation 6 respectively.

$$\begin{aligned} O_1 &= A1_1 \cdots A2_1 \cdot An_1 \\ O_0 &= A1_0 \cdots A2_0 \cdot An_0 \end{aligned} \tag{5}$$

$$\begin{aligned} O_1 &= A1_1 + \cdots + A2_1 + An_1 \\ O_0 &= A1_0 + \cdots + A2_0 + An_0 \end{aligned} \tag{6}$$

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 Equation 7 and Equation 8 respectively.

$$\begin{aligned} O_1 &= \overline{A1_0 \cdots A2_0 \cdot An_0} \\ O_0 &= \overline{A1_1 \cdots A2_1 \cdot An_1} \end{aligned} \tag{7}$$

$$\begin{aligned} O_1 &= \overline{A1_0 + \cdots + A2_0 + An_0} \\ O_0 &= \overline{A1_1 + \cdots + A2_1 + An_1} \end{aligned} \tag{8}$$

With the GLIFT logic for the AND, OR and NOT gates, one can constructively generate GLIFT logic for any Boolean circuit. The next subsection first gives an insight into the advantages of both encoding techniques when targeting different application scenarios and then defines encoding and decoding logic needed to convert between the two techniques.

C. Encoding and Decoding Logic

As will be shown with experimental results, the new encoding can be more efficient in modeling taint propagation. However, it is not as efficient as the old one in data storage and transmission. This is because tainted '1' and tainted '0' share the same code "01". An additional bit is needed to distinguish the two, which will consume extra memory and communication bandwidth. In order to take advantage of both encodings, there needs to be encoding and decoding logic that perform the conversion between different encodings.

For input A and its taint a_t (old encoding), the following encoding logic is needed to convert them to the new encoding, which is denoted by A_1 and A_0.

$$\begin{aligned} A_1 &= A \cdot \overline{a_t} \\ A_0 &= A + a_t \end{aligned} \tag{9}$$

At the output, decoding logic is needed to determine the taint o_t (old encoding) upon O_1 and O_0, which are the taint outputs in the new encoding. This is achieved through a single exclusive OR operation as shown in Equation 10.

$$o_t = O_1 \oplus O_0 \tag{10}$$

The encoding and decoding logic are linear to the number of I/Os of a design. They only need to be deployed in the top level entity. Thus, they will not introduce significant overheads in area and delay as compared to the taint propagation logic. With the encoding and decoding logic, one can perform conversions between the two encoding schemes. The following subsection carries out a comparison of GLIFT logic represented in both encodings.

D. Comparison to the Old Encoding Technique

Figure 2 (a)∼(c) and (d)∼(h) show the GLIFT logic for basic gates represented in the old and new encoding techniques respectively.

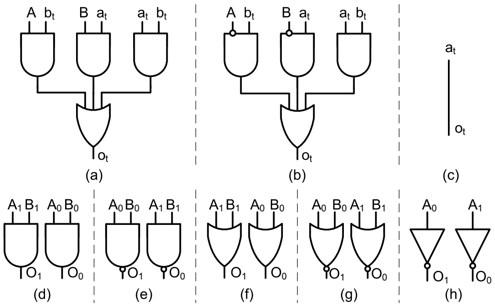


Fig. 2. (a)∼(c) GLIFT logic for AND/NAND, OR/NOR and NOT gate using the old encoding technique. (d)∼(h) GLIFT logic for AND, NAND, OR, NOR, and NOT gate using the new encoding technique.

To represent the GLIFT logic for the NOT gate, the old encoding technique simply uses a wire as shown in Figure 2 (c) while the new encoding uses two inverters as shown in Figure 2 (h). The new GLIFT logic tends to report larger area and delay if the original design consists of more NOT gates than AND and OR gates.

From Figure 2 (d) to (g), we can see that the new GLIFT logic only introduces an additional gate without changing the number of logic levels. In the best case, the GLIFT logic circuit represented in the new encoding technique will be twice of the original circuit in area and equal in delay if there is no NOT gate in the design. It will be half in area and delay on average when compared to the GLIFT logic represented in the old encoding technique. Such a conclusion can be drawn by comparing Figure 2 (a)∼(b) and (d)∼(g) and will be reinforced in the results section.

As given in Equations 5 to 8, the GLIFT logic for n-input AND, OR, NAND and NOR gates are 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 Equations 2 and 3. Additionally, the GLIFT logic represented in the new encoding technique has a constant one logic level even for a large n. By comparison, logic levels in the GLIFT logic for AND and OR represented using the old encoding technique increases linearly to n according to Equations 2 and 3.

Besides the AND, OR, NAND and NOR gates, the old GLIFT logic for more complex Boolean gates are also more complicated. For a better understanding, let's consider the two-input multiplexer (MUX-2), whose logic equation is $O = AB + \overline{B}C$. 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 generate tracking logic for them step by step. Under the new encoding, this process is more straightforward. One can directly write out the new GLIFT logic for MUX-2 as shown in Equation 11. Further, tracking logic can be easily created for large components as long as their logic functions are specified in sum-of-products or product-of-sums, or more generally in a form in which all inverse operations are applied to logic variables only.

$$\begin{aligned} O_1 &= A_1B_1 + \overline{B_0}C_1 \\ O_0 &= A_0B_0 + \overline{B_1}C_0 \end{aligned} \tag{11}$$

Further, by observing a partial logic circuit such as shown in Figure 3, one can discover another difference between the two encoding techniques. In the old encoding technique, the original logic variables and intermediate results (the wires with shadow) are referenced by the GLIFT logic, which results in a nested design of the GLIFT logic and original circuit. In the new technique, the encoding results contain all the information necessary for taint propagation. Thus, taint is propagated independent of the original circuit and GLIFT logic is separated from the original design.

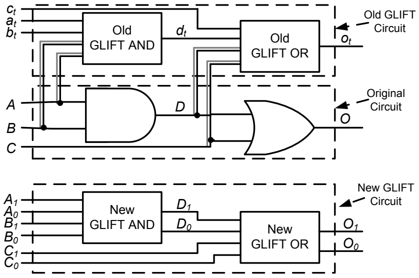


Fig. 3. Partial circuit with GLIFT logic represented in both encodings. The old GLIFT circuit needs intermediate results from the original design. The new GLIFT circuit is completely independent from the original circuit.

Now that the GLIFT logic is separated from the original circuit, it can be designed and verified independently. As a result, the design complexity is decreased and the design process can be parallelized. In addition, such separation is quite useful for a static verification scenario, in which the GLIFT logic is only used to verify if the original circuit under test complies with pre-defined information flow policies and will be removed when the verification completes. Further, such separation also facilitates the 3-D integration of GLIFT logic as a stacked security layer as described in [14] because communication between the security layer and the original design is significantly reduced.

Apart from separation from the original design, the new GLIFT logic can also be configured as redundancy for fault tolerance. This will be covered in the next subsection.

E. Enabling Circuit Redundancy

Another highlight of the new encoding is that it enables GLIFT logic to function as circuit redundancy when no input is tainted. Such highlight originates from the observation that when no input is tainted, the new GLIFT logic will behave exactly the same as the original design.

According to Figure 2 (d), (f) and (h), the GLIFT logic for the basic constructs used in the constructive method, i.e., AND, OR and NOT, are two AND, OR and NOT gates correspondingly. In Equation 9, when $a_t = 0$, both A_1 and A_0 will take the value of A. Similarly, all inputs to the new GLIFT circuit will take values of their original variables when they are not tainted. Thus, in the case when no input is tainted, the new GLIFT logic will be just twice the original design and it will function as circuit redundancy.

For a better understanding, let's consider the original circuit and GLIFT logic for a two-input multiplexer as shown in Figure 4. When no input is tainted ($a_t = 0, b_t = 0$ and $c_t = 0$ in the old encoding), we have $A_1 = A_0 = A$, $B_1 = B_0 = B$ and $C_1 = C_0 = C$. When these values are propagated through the GLIFT logic, the same values will be observed at the outputs of both the original circuit and the GLIFT logic, i.e., $O_1 = O_0 = O$. In other words, GLIFT logic acts as redundancy to the original design.

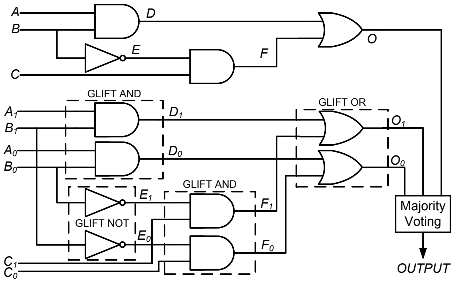


Fig. 4. The original circuit and GLIFT logic for a two-input multiplexer. The outputs O_1 and O_0 will be equal to O when no input is tainted. In this case, the new GLIFT logic functions as circuit redundancy.

This highlight of the new encoding technique can be quite useful because high-assurance systems usually require circuit redundancy for fault tolerance. One can ground all the taints (setting all the taints in the old encoding to zero) and check if the original circuit and the GLIFT logic have identical outputs for fault detection. It implements triple modular redundancy (TMR) [15] since there are two redundant bits for each output. If a fault is detected in the original circuit, the GLIFT logic can be temporarily used as substitute of the original design.

Now that a comparison of efficiency has been performed theoretically, the following section presents a comparison of the two encoding techniques in terms of area, delay and simulation time using *IWLS* benchmarks.

IV. Experimental Results

Apart from being able to be configured as circuit redundancy, GLIFT logic circuits represented in the new encoding technique achieve significant reductions in area, delay and simulation time. In subsection IV-A, GLIFT logic for *IWLS* benchmarks are generated using both encoding techniques for area and delay analysis. Subsequently, simulation time results are provided in subsection IV-B.

A. Area and Delay Results

Area and delay are important issues that need to be taken into consideration in GLIFT logic design if GLIFT is to be fabricated for dynamic information flow tracking. We carried out experiments on several *IWLS* benchmarks to obtain area and delay reports for GLIFT logic circuits represented in both encoding techniques. *ABC* [16] is used as the synthesis tool. The *resyn2* synthesis script in *ABC* is used to optimize the GLIFT logic circuits because it provides a good tradeoff between area and delay [16]. The *mcnc* library embedded in *ABC* does not provide units for area and delay results. Table V shows some statistics of the original benchmarks used in our experiments including the number of I/O, number of gates, area and delay.

TABLE V
Statistics of benchmarks used for area and delay analysis.

Benchmark	# of I/O	# of Gates	Area	Delay
ttt2	24/21	158	384	6.5
alu2	10/6	678	1737	11.0
alu4	14/8	3569	9264	14.4
vda	17/39	1210	2973	10.6
x1	51/35	483	1167	10.4
i5	133/66	253	515	9.0
i6	138/69	437	983	4.6
i7	199/67	501	1151	5.3
i8	133/81	28441	72544	13.8
i9	88/63	10717	27152	11.9
frg2	143/139	4748	11597	14.1
too_large	38/3	6568	16819	18.6
t481	16/1	515	1270	14.0

In our experiment, both encoding techniques are used for GLIFT logic representation. The GLIFT logic circuits are then synthesized for area and delay reports, which are shown in Table VI. The area results include both the original benchmark and its GLIFT logic. *Reductions* in area and delay achieved by the new encoding technique are given in percentage.

From Table VI, we can discover that GLIFT logic represented in the new encoding technique gives significantly smaller area and delay. As an example, the GLIFT logic for *alu4* described using the old encoding technique reports an area/delay of 52893/47.4, while that represented using the new one reports a result of 27791/17; there are 47.5% reduction in area and 64.1% reduction in delay.

The new encoding technique has also achieved significant reductions in area and delay when applied to the remaining benchmarks in the *IWLS* benchmark set. On average, the GLIFT logic represented in the new encoding technique decreases the area by 25.7%, delay by 31.4% and area-delay production by 53.5% on the 30 largest benchmarks in the complete set. The improvement in area results from simpler GLIFT logic for basic logic gates, especially for large AND and OR gates. In addition, the new encoding technique also reduces the number of logic levels and thus achieves significantly smaller delay results.

TABLE VI
Area/delay results of logic circuits (both original benchmark and its GLIFT logic).

Benchmark	Area			Delay		
	OldEnc.	NewEnc.	Reduc.	OldEnc.	NewEnc.	Reduc.
ttt2	2988	1324	55.7%	21.7	9.2	57.6%
alu2	8518	5025	41.0%	32.8	13.9	57.6%
alu4	52893	27791	47.5%	47.4	17.0	64.1%
vda	14742	9041	38.7%	34.9	13.1	62.5%
x1	8875	4257	52.0%	37.0	13.0	64.9%
i5	6461	2395	62.9%	16.5	10.5	36.4%
i6	5379	3679	31.6%	9.3	7.7	17.2%
i7	8819	5886	33.3%	12.0	8.4	30.0%
i8	400907	206727	48.4%	47.5	16.8	64.6%
i9	159116	77677	51.2%	35.1	14.5	58.7%
frg2	162214	38050	76.5%	45.0	17.0	62.6%
too_large	182469	48230	73.6%	63.1	21.4	66.1%
t481	11115	4920	55.7%	44.1	17.8	59.6%
Average			51.4%			54.0%

Area and delay overheads are major concerns when deploying GLIFT logic in a dynamic application scenario. Table VII lists overheads of new GLIFT logic circuits in terms of area and delay. The area results for GLIFT logic include both the original benchmark and tracking logic.

TABLE VII
Area/delay overheads of new GLIFT logic circuits (original benchmark included in GLIFT logic).

Benchmark	Area			Delay		
	Orig.	GLIFT	Overhead	Orig.	GLIFT	Overhead
ttt2	384	1324	3.45	6.5	9.2	1.42
alu2	1737	5025	2.89	11.0	13.9	1.26
alu4	9264	27791	3.00	14.4	17.0	1.18
vda	2973	9041	3.04	10.6	13.1	1.24
x1	1167	4257	3.65	10.4	13.0	1.25
i5	515	2395	4.65	9.0	10.5	1.17
i6	983	3679	3.74	4.6	7.7	1.67
i7	1151	5886	5.11	5.3	8.4	1.58
i8	72544	206727	2.85	13.8	16.8	1.22
i9	27152	77677	2.86	11.9	14.5	1.22
frg2	11597	38050	3.28	14.1	17.0	1.21
too_large	16819	48230	2.87	18.6	21.4	1.15
t481	1270	4920	3.87	14.0	17.8	1.27
Average			3.48			1.30

From Table VII, we can discover that the area of the GLIFT logic is about twice the original design (after removing the original circuit) and the delay is comparable to the original design. These results well agree with what are shown in Figure 2 and are reinforcements of discussions in Section III-D.

B. Simulation Time Results

From area and delay analysis, we can see that the GLIFT logic usually dominates the original design due to its complexity. Consequently, it takes a long time to verify a design that integrates its GLIFT logic. Simulation time is another important factor that should be taken into consideration.

Several *IWLS* benchmarks are used for simulation time analysis. GLIFT logic represented in both encoding techniques are simulated under Modelsim 6.4a using LFSR (Linear Feedback Shift Register) as the random source. For each benchmark, we simulated until a coverage of 95% is reached. In the simulation, a total number of 2^{22} vectors are tested, which meets the simulation coverage requirement. The simulation time results are shown in Figure 5. The data over the bars are simulation time reductions in percentage.

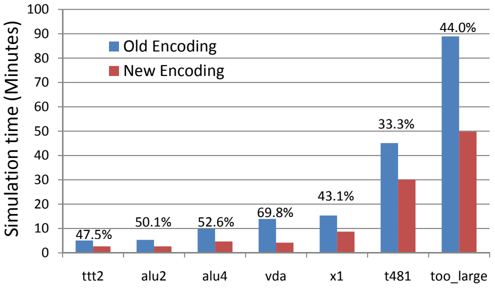


Fig. 5. Simulation time for GLIFT logic represented in different encoding techniques.

From Figure 5, the simulation time of GLIFT logic represented in the new encoding technique is on average half of tracking logic represented in the old encoding as shown by the percentage data. One of the reasons for such an improvement is that the GLIFT logic for n-input AND and OR gates represented by the new encoding technique is constantly two product terms whose size is linear n while that represented using the old technique has a $2^n - 1$ equal size product terms. The other reason which contributes to shorter simulation time is that the new encoding technique reduces the number of logic levels in the GLIFT logic, which is discussed in Section III-D.

From the experimental results, we can see that the new encoding technique is far more efficient in describing taint propagation for GLIFT because of the significant reductions in area and delay. These improvements are essential when GLIFT is to be deployed into hardware for high-assurance systems since overhead of the existing GLIFT logic is too large to be practical to go in a fabricated chip. The reduction in simulation time will speed up circuit testing even if GLIFT is only used to statically verify if a circuit complies with predefined information flow policies.

Theoretically, GLIFT logic represented using both encoding techniques can potentially be optimized to the same circuit after encoding and decoding logic are inserted. Equivalence check using the *cec* command integrated in the *ABC* tool has further enforced such theoretical analysis. However, logic optimization is a hard problem and GLIFT logic is far more complex than the original circuit. Reductions in product term count and the number of logic level in GLIFT logic represented in the new encoding technique result in far better implementation results and also shorter simulation time.

V. Conclusion

GLIFT provides an effective way to understand all information flows and prove important security properties of a system. Generating optimized GLIFT logic circuits is a critical issue due to their large overheads in area and delay. This article proposes an improved encoding technique for GLIFT logic representation. The proposed encoding technique is functionally equivalent to the existing one in modeling taint propagation but simplifies the GLIFT logic for large AND, NAND, OR and NOR gates. The new encoding technique separates the GLIFT tracking logic from the original design and enables the GLIFT logic to be configured as circuit redundancy for fault tolerance. Experimental results have also shown that the proposed encoding technique decreases the area, delay and simulation time of GLIFT logic circuits by 20% to 60% as compared to the old encoding technique.

References

[1] J. A. Goguen and J. Meseguer, “Security policies and security models,” *IEEE Symposium on Security and Privacy*, pp. 11-20, April 1982.

[2] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[3] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *Journal of Computer Security*, 4(2-3):167-187, Jul. 1996.

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

[5] M. Krohn, A. Yip and M. Brodsky et al., “Information flow control for standard os abstractions,” *ACM SIGOPS Operating Systems Review (SOSP ’07)*, 41(6):321-334, 2007.

[6] M. Dalton, H. Kannan, and C. Kozyrakis, “Raksha: A Flexible Information Flow Architecture for Software Security,” In *34th Intl. Symposium on Computer Architecture (ISCA)*, pp. 482-493. June 2007.

[7] G. Venkataramani, I. Doudalis, Y. Solihin and M. Prvulovic, “FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation,” in *Proc. of the 14th International Symposium on High-Performance Computer Architecture (HPCA-14)*, pp. 173-184, February 2008.

[8] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, “Complete information flow tracking from the gates up,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 109-120, 2009.

[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*, pages 225-242. Springer-Verlag, 2007.

[11] W. M. Hu, “Reducing Timing Channels with Fuzzy Time,” In *Proceedings of the Symposium on Research in Security and Privacy*, pp. 8-20, May 1991.

[12] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood and R. Kastner, “Theoretical Analysis of Gate Level Information Flow Tracking,” in *Proceedings of the Design Automation Conference (DAC)*, pp. 244-247, June 2010.

[13] K.J. Adams, J.G. Campbell, L.P. Maguire and J.A.C. Webb, “State assignment techniques in multiple-valued logic,” in *Proceedings of the 29th IEEE International Symposium on Multiple-Valued Logic*, pp. 220-225, May 1999.

[14] T. Huffmire, J. Valamehr, T. Sherwood and R. Kastner, et al., “Extended abstract: Trustworthy system security through 3-D integrated hardware,” *IEEE International Workshop on Hardware-Oriented Security and Trust, 2008 (HOST 2008)*, pp. 91-92, June 2008.

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

[16] Berkeley Logic Synthesis and Verification Group, *ABC: A System for Sequential Synthesis and Verification*, Release 70930. http://www.eecs.berkeley.edu/∼alanmi/abc/