Theoretical Fundamentals of Gate Level Information Flow Tracking

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*—Information flow tracking is an effective tool in computer security for detecting unintended information flows. However, software based information flow tracking implementations have drawbacks in preciseness and performance. As a result, researchers have begun to explore tracking information flow in hardware, and more specifically, understanding the interference of individual bits of information through logical functions. Such gate level information flow tracking (GLIFT) can track information flow in a system at the granularity of individual bits. However, the theoretical basis for GLIFT, which is essential to its adoption in real applications, has never been thoroughly studied. This paper provides fundamental analysis of GLIFT by introducing definitions, properties, and the imprecision problem with a commonly used shadow logic generation method. This paper also presents a solution to this imprecision problem and provides results that show this impreciseness can be tolerated for the benefit of lower area and delay.**

***Index Terms*—Gate level information flow tracking, hardware, information flow tracking, security.**

I. Introduction

HIGH assurance systems such as flight control networks and financial, military and medical systems, require strict guarantees on correct operation or face catastrophic consequences. Two common policies that need to be upheld in these systems are non-interference [1] and the Bell LaPadula confidentiality policy [2]. The non-interference policy enforces that an untrusted subsystem should never influence a trusted one, e.g., data from passenger network shall never affect the flight control network on an airplane. The Bell LaPadula confidentiality policy requires that information from a classified subsystems does not leak to an unclassified one, e.g., medical care records should not be observed from an open network. There are many different ways to enforce non-interference and confidentiality, such as physical isolation, access control, and information flow tracking (IFT). IFT is a frequently used technique due to its efficiency in detecting software attacks and preventing unexpected information flows.

Manuscript received June 16, 2010; revised November 21, 2010; accepted January 27, 2011. This work was supported by the NSF, under Grant CNS-0910581. This paper was recommended by Associate Editor V. Bertacco.

W. Hu is with the School of Automation, Northwestern Polytechnical University, Xi'an, Shaanxi 710072, China, and also with the Department of Computer Science and Engineering, University of California, San Diego, CA 92093 USA (e-mail: vinnie@mail.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 and T. Sherwood are with the Department of Computer Science, University of California, Santa Barbara, CA 93106 USA (e-mail: tiwari@cs.ucsb.edu; sherwood@cs.ucsb.edu).

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

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/TCAD.2011.2120970

Plenty of work has been done in IFT in programming languages. A survey paper by Sabelfield and Myers [3] discusses a large number of current language-based IFT methods. Other software IFT approaches focus on OS level security policies and tracks information flow with processes [4], [5]. However, software based IFT typically has drawbacks in design complexity, performance and precision since these methods are lack of understanding of the underlying hardware. As a result, recently IFT has been implemented in hardware to take a more bottom-up approach to security. Many of these hardware IFT designs target the instruction set architectures (ISA) [6], [18]. We observe that these ISA level hardware-based approaches tend to be overly conservative, in that they indicate that there were unintended information flows in the system when there was in fact none because they usually use coarse granularity labels and propagation policies.

To track information flow more precisely, we proposed gate level information flow tracking (GLIFT) [7]. GLIFT propagates information at a very fine granularity by tracking each bit in a system. To obtain this level of precise information flow, the original logic circuit under test is combined with *tracking* or *shadow* logic. This extra logic can be fabricated and used at runtime or instantiated in the design phase to verify whether the hardware design enforces the desired information flow policies. The method we used in [7] *constructively* creates shadow logic for each gate in a system. It implements shadow logic for each gate in the system discretely. However, for certain logic functions, this shadow logic generation method is overly conservative as well; the shadow logic would indicate that there was information flow when there was none.

This paper presents a theoretical basis for GLIFT by introducing essential definitions, properties, and various shadow logic generation methods. It also provides a solution to the problem of a commonly used shadow logic generation method being overly conservative. The major contributions of this paper are as follows.

1) *Definitions and theoretical proofs of fundamental properties of GLIFT:* We present essential definitions, prove fundamental properties of shadow logic, and propose a

symbolic representation and formal analysis of shadow logic for common logical constructs (AND, OR, NOT, NAND, NOR, XOR).

2) *Preciseness of shadow logic:* We show the problem associated with imprecise information flow tracking and formally prove that a logic function with all prime implicants produces precise shadow logic when shadowed constructively.
3) *Analytic and quantitative analysis of shadow logic:* We present quantitative analysis of the shadow logic for basic Boolean functions and *ISCAS* and *IWLS* benchmarks, by comparing the area, delay, and preciseness of shadow logic circuits generated with different methods.

The remainder of this paper is organized as follows. Section II covers the related work in IFT. In Section III, we formally define basic concepts of GLIFT, prove fundamental properties, and propose a symbolic representation as well as quantitative analysis of the shadow logic for common logical constructs. Section IV introduces various shadow logic generation methods with an analysis of their computational complexity. We show the potential impreciseness of shadow logic, provide a possible solution, and formally prove it to be effective in Section V. Section VI provides complexity results in minterm counts and implementation results of different shadow logic implementations in terms of area, delay as well as preciseness using *ISCAS* and *IWLS* benchmarks. We conclude and present future work in Section VII.

II. Information Flow Tracking

Information flow tracking is used to prevent secure data from leaking to public entities or malicious information from affecting protected data. IFT can be implemented either statically, by monitoring where information could flow to verify if this violates a specified security policy, or dynamically, by assigning data with a tag then observing where this tag propagates through the system [10]. This section discusses various implementations of IFT.

Denning [11] was first to study using static analysis to enforce information flow policies with little run-time overhead. Type-based systems proposed by Volpano *et al.* [12] and Pottier *et al.* [13] provide another way to track information flow security through static compile-time analysis to prevent information flows from a labeled trusted type to an untrusted one. A typical example of such type checking systems is the Jif compiler [14]. However, security policies in these static approaches need to be defined prior to execution. This forces programmers to comply with a new typing system. The drawbacks of static IFT schemes in flexibility and complexity motivate researchers to track information dynamically.

Some dynamic schemes of IFT can be implemented at the operating system and application levels by monitoring information flows with processes. Asbestos [4] is an operating system that uses labels associated with each process to determine what operations a process can perform and which processes it can interact with. Flume [5] provides a user-level reference monitor on Linux that restricts untrusted processes from invoking system calls directly. Gupta *et al.* [16] proposed scalable dynamic information flow tracking techniques under multithread schemes. Lewis and Sturton [17] implemented distributed protocols for IFT on a multi-core architecture using Asbestos operating system style labels [4] and message passing. However, these approaches tend to introduce memory and delay overheads which hinder the system's performance. All of these designs tend to track information at a very coarse granularity and designs that precisely track all information flows are needed.

More precise approaches track information in hardware. Dynamic information flow tracking (DIFT), proposed by Suh *et al.* [6], tags information from untrusted channels and tracks it throughout a processor. Raksha [18], FlexiTaint [19], and Flexible HW [20] are typical DIFT systems. These systems tend to track information at the instruction or word level and propagate labels in a conservative manner. In other words, if either operand in an operation is tagged, then the result is tagged. GLIFT [7] provides a more precise approach by tracking information at the fine granularity of bits. It is flexible enough to be applied to any digital hardware design and is not centralized around micro-architectures since the method targets gates and not micro-architectural units, e.g., multiplexer, function unit, and register file.

GLIFT targets secure systems and can be used to prove both the confidentiality of sensitive data as well as integrity of the system. For confidentiality, GLIFT can detect whether secret inputs are leaking to unclassified outputs. An example of violations in confidentiality can be seen in cache timing attacks, where a secret key is leaked through timing channels elicited by the cache hit/miss latencies [21]. It can also be used to detect integrity violations where untrusted inputs flow to trusted outputs. An example of system integrity can be found in the Boeing 787 aircraft, which is equipped with connectivity between the user and flight control networks [22]. It is important to show that these systems are isolated meaning that actions taken on the user network do not affect the flight control network.

In previous work, we used GLIFT to develop an air-tight information flow tracking microprocessor that tracks all information flows emanating from untrusted inputs [7]. We improved upon this architecture to allow regions of execution to be tightly quarantined and their side effects to be tightly bounded [8], which again employs GLIFT to track information flows. Preliminary work [9] of this paper has provided a theoretical foundation of GLIFT by defining and proving its fundamental properties and formalized a symbolic representation of the tracking logic for Boolean gates. This paper performs further theoretical analysis by introducing various tracking logic generation methods and discussing the imprecision problem with GLIFT. It also presents quantitative analysis results in term of area, delay, preciseness and computational complexity.

To better understand GLIFT tracking logic, the following section provides a detailed discussion about the definition and properties of GLIFT.

III. Concepts and Fundamental Properties of GLIFT

GLIFT provides an effective approach to track information flow by labeling each data bit with a tag and tracking the prop-

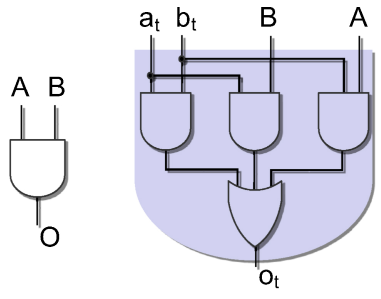


Fig. 1. (a) Two-input AND gate. (b) GLIFT tracking logic of two-input AND gate is $Ab_t + Ba_t + a_tb_t$. Every change at the input of the gate is precisely tracked at the output.

agation of this bit through separate tracking logic. This section first provides some terms and definitions of GLIFT, proves fundamental properties, proposes a symbolic representation of the shadow logic for primitive gates and finally analyzes the GLIFT characteristics of logical constructs.

Throughout this paper, upper-case letters without/with a subscript such as $A, B, A_i\,(i = 1, 2, \cdots, n)$ are used to denote logic variables and lower-case letters with a subscript, a_t, b_t, a_i, are used to denote the *taint* of A, B, A_i, respectively. f, g, and h are used to denote logic functions; $sh(f)$, $sh(g)$, and $sh(h)$ are used to denote the shadow logic functions for f, g, and h. The following subsection formally defines the concepts of taint, original logic function, shadow logic function and preciseness of shadow logic function.

A. Terms and Definitions

Definition 1—taint: Taint is a tag associated with a data bit indicating that this bit should be tracked through the system. Taint is propagated from the input to the output of a function if the tainted input has an influence on the output. A logic variable is said to be tainted when its taint is logic true. For a better understanding of taint, consider the two-input AND gate (AND-2) in Fig. 1(a). To see if the tainted inputs can affect the output, we toggle the tainted inputs and look at the output. If a change at the output is observed by changing the tainted inputs, the output is marked as tainted.

Table I is the truth table of AND-2 with taint information, where a_t and b_t are the taints of inputs A and B, respectively. Let us first consider *row*7 in the truth table ($A = 0$, $B = 1$, $a_t = 1$, $b_t = 0$). When changing the value of the tainted input A, the output will change. Thus, the output should be marked as tainted ($o_t = 1$). Then let us consider *row*4 ($A = 0$, $B = 0$, $a_t = 1$, $b_t = 1$), which has two tainted inputs. When changing the value of either tainted inputs, i.e., A or B, the output does not change. However, when the values of both tainted inputs are changed, a change in the output will be observed. Thus, the output should also be marked as tainted ($o_t = 1$).

It is important to notice that whenever one input of AND-2 is untainted 0, the output will be untainted regardless of the other input. Examples of such cases can be found in *row*3 and *row*6 in the truth table. In these cases, untainted 0 prevents tainted information from flowing to the output. Since those entries with an untainted 0 are not tracked by the shadow logic, GLIFT more precisely covers the truth table than previous hardware IFT implementations. When the logic circuit specified by the truth table is simplified, the shadow logic for AND-2 is obtained as shown in Fig. 1(b).

TABLE I

TRUTH TABLE OF TWO-INPUT AND GATE WITH TAINT INFORMATION

#	A	B	a_t	b_t	O	o_t
1:	0	0	0	0	0	0
2:	0	0	0	1	0	0
3:	0	0	1	0	0	0
4:	**0**	**0**	**1**	**1**	**0**	**1**
5:	0	1	0	0	0	0
6:	0	1	0	1	0	0
7:	**0**	**1**	**1**	**0**	**0**	**1**
8:	0	1	1	1	0	1
9:	1	0	0	0	0	0
10:	1	0	0	1	0	1
11:	1	0	1	0	0	0
12:	1	0	1	1	0	1
13:	1	1	0	0	1	0
14:	1	1	0	1	1	1
15:	1	1	1	0	1	1
16:	1	1	1	1	1	1

In Fig. 1(b) it can be seen that if both inputs are tainted, then tainted information is clearly flowing to the output. The more subtle cases are when a single input is tainted. In these cases, the output will be tainted if either B is tainted and A is logic 1 or when A is tainted and B is logic 1. In both of these cases, a change in the tainted input will result in a change at the output. Thus, in these situations, information from the tainted input flows to the output of the gate. This is unlike previous conservative approaches in which the output is said to be tainted when *any* inputs are tainted.

Definition 2—original logic function: The original logic function f is the function under test. This function is given the following form, where $A_1, A_2, \cdots, A_n, \overline{A_1}, \overline{A_2}, \cdots, \overline{A_n}$ are the inputs and their complements

$$f = f_n(A_1, A_2, \cdots, A_n, \overline{A_1}, \overline{A_2}, \cdots, \overline{A_n}).$$

The original logic function is tainted when there exists at least one subset of the tainted inputs such that a change at the output is observed by changing the values of these tainted inputs. In other words, the tainted inputs can affect the result of the output.

Definition 3—shadow logic function: The shadow logic function of an original logic function indicates the taint status of that original logic function. It will output logic true whenever a tainted input can affect the output of the original logic function and logic false when there is no tainted information flow through the original logic function. The shadow logic function for a given function f, denoted by $sh(f)$, is a function of the original logic function variables, their complements and taints, given in the form of

$$sh(f) = sh(A_1, A_2, \cdots, A_n, \overline{A_1}, \overline{A_2}, \cdots, \overline{A_n}, a_1, a_2, \cdots, a_n).$$

Definition 4—preciseness of shadow logic function: A shadow logic function is said to be precise if it indicates logic true *iff* tainted information flows from the input to the output in the original logic function, otherwise it is said to be conservative. In other words, a precise shadow logic function indicates logic true when and only when at least one tainted input has an influence on the output in the original logic function. A precise shadow logic function will contain the minimum number of minterms among all shadow logic functions that safely track all the information flows.

With an understanding of essential definitions of GLIFT, we can now discuss fundamental properties of GLIFT through theoretical proofs.

B. Fundamental Properties of GLIFT

GLIFT has some fundamental properties that can be used to derive shadow logic. They are formally stated below as theorems and lemmas.

Theorem 1: There exists a shadow logic function for a given original logic function that contains only one of either A_i, $\overline{A_i}$, or a_i in any product term.

Proof 1: Only tainted logic variables have a chance to taint the original logic function. Thus, there exists a shadow logic function that contains only the taints of the logic variables but not their inverse. Meaning, terms containing $\overline{a_i}$ $(i = 1, 2, \cdots, n)$ will not be present in such a shadow logic function.

If the product of A_i and a_i $(i = 1, 2, \cdots, n)$ appears in a shadow logic function, there must be at least one other product term that contains the product of $\overline{A_i}$ and a_i among all the terms of the shadow logic function in order for simplification to occur.

Assume that the following term appears in a shadow logic function:

$$m_1(B_1, B_2, \cdots, \mathbf{A_i = 1}, \cdots, B_r, b_1, b_2, \cdots, a_i = 1, \cdots, b_r)$$
$$B_j \in \{0, 1\}, b_j \in \{0, 1\}, j = 1, 2, \cdots, r.$$

By the definition of taint, changing a tainted input needs to be tracked by the shadow logic function, thus another term

$$m_2(B_1, B_2, \cdots, \mathbf{A_i = 0}, \cdots, B_r, b_1, b_2, \cdots, a_i = 1, \cdots, b_r)$$
$$B_j \in \{0, 1\}, b_j \in \{0, 1\}, j = 1, 2, \cdots, r$$

will also taint the output since the logic values and taints of the rest logic variables are kept unchanged. As a consequence, terms with products of A_i and a_i always appear in pairs (e.g., $m_1 + m_2$) in the shadow logic function. This allows sub-terms m_1 and m_2 to be eliminated. Thus, a shadow logic function can always be described in a form that contains only one of either A_i, $\overline{A_i}$, or a_i in any product term. ■

Lemma 1.1: A logic variable A can be eliminated from a product term that contains its taint a_t.

According to Theorem 1, terms with product of Aa_t and $\overline{A}a_t$ always appear in pairs. Thus, such terms can be expanded by eliminating the logic variable.

Theorem 2: The shadow logic function of a single variable function is the taint of that logic variable.

Proof 2: The shadow logic function of a single variable function should contain the taint of that logic variable in all product terms to indicate tainted information flow. According to Theorem 1, there exists a shadow logic function in which the logic variable and its inverse never appear since all product terms already contain the taint. Thus, it should be exactly the taint of that logic variable. ■

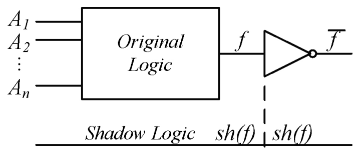


Fig. 2. Inverter changes the value of the logic input but does not affect the taint bit.

Lemma 2.1: An inverter changes the value of the logic input but does not affect the taint bit.

As mentioned, all changes to tainted inputs of a function need to be tracked in order to correctly track all information flows. So single variable logic functions such as $g_1 = A$ and $g_2 = \overline{A}$ share the same shadow logic function $sh(g_1) = sh(g_2)$. In other words, the taint bit propagates from input to output of an inverter regardless of the original logic function's inputs.

Lemma 2.2: Inverting the original logic function does not invert the shadow logic function.

Fig. 2 shows a black-boxed logical function with an inverter at the output. As already mentioned, the shadow logic function for the inverse of a variable is equal to the shadow logic function of the variable itself. Thus, inverting a function does nothing to its shadow logic. In other words, for any logic function f, $sh(f) = sh(\overline{f})$.

Now that some fundamental properties of GLIFT are proved, the next subsections discuss the shadow logic functions for logic gate primitives by presenting a formal representation and minterm count analysis.

C. Deriving Shadow Logic Function for Logic Primitives

1) *Representing AND Expressions:* The general form of a logic AND expression is $f = g \cdot h$, where g and h can be logic variables or functions. Assume the shadow logic functions for terms g and h have already been realized and use $sh(f)$, $sh(g)$ and $sh(h)$ to denote the shadow logic function of f, g, and h, respectively. Referring the shadow logic function for AND-2 as shown in Fig. 1(b), the resulting shadow logic function is shown in

$$sh(f) = g \cdot sh(h) + h \cdot sh(g) + sh(g) \cdot sh(h). \tag{1}$$

To conceptually understand (1), it can be seen that the first term on the right side of the formula means when g is untainted and logic true, $sh(h)$ will determine if f is tainted. When h is untainted and logic true, $sh(g)$ will determine the taint. The final term shows that if both g and h are tainted, then the result should be tainted. The above formula can be given a different form as shown below in (2). This representation is symbolic

$$sh(f) = (g + sh(g)) \cdot (h + sh(h)) - g \cdot h. \tag{2}$$

Although (2 is symbolic, it provides a good conceptual understanding of taint. $g + sh(g)$ means two conditions of g contribute to an tainted output, i.e., untainted true or tainted. h makes a similar contribution to the shadow logic function here. The minus sign means removing the term $g \cdot h$ from the expression because it never affects the output of the shadow logic function.

The same methodology can be used for a three-input AND gate (AND-3). Assume $f = g \cdot h \cdot k$. Using (2), the result below is obtained

$$\begin{aligned} sh(f) &= (g + sh(g)) \cdot (h \cdot k + sh(h \cdot k)) - g \cdot h \cdot k \\ &= (g + sh(g)) \cdot (h + sh(h)) \cdot (k + sh(k)) - g \cdot h \cdot k. \end{aligned} \tag{3}$$

With the minus operation, which means removing the term that follows from the equation, a general form of the n-input AND gate (AND-N) shadow logic function can be given as (4) (multiplication is defined for logic function with meaning of logic AND operation)

$$sh(f = f_1 \cdot f_2 \cdots f_n) = \prod_{i=1}^{n} (f_i + sh(f_i)) - f. \tag{4}$$

Additionally, the shadow logic function for n-input NAND gate is identical to (4) by Lemma 2.1.

2) *Representing OR Expressions:* The general form of a logic OR expression is $f = g + h$, where g and h can be logic variables or functions. As before $sh(f)$, $sh(g)$, and $sh(h)$ represent the shadow logic functions of f, g, and h, respectively. By Lemma 2.1, $sh(f) = sh(\overline{f})$ is true for any logic function. So the result for the AND-2 expression can be used to obtain the shadow logic for OR-2 (two-input OR gate) directly. When rewriting the logic function for OR-2 using De Morgan's Law, the following holds:

$$f = \overline{\overline{g} \cdot \overline{h}}. \tag{5}$$

Lemma 2.1 states that $\overline{f} = \overline{g} \cdot \overline{h}$ has the same shadow logic for a function $f = \overline{g} \cdot \overline{h}$. Thus, the same approach can be used for OR-2 as it was for AND-2

$$sh(f) = sh(\overline{f}) = \overline{g} \cdot sh(\overline{h}) + \overline{h} \cdot sh(\overline{g}) + sh(\overline{g}) \cdot sh(\overline{h}). \tag{6}$$

Because $sh(\overline{g}) = sh(g)$ and $sh(\overline{h}) = sh(h)$ by Lemma 2.1, the result is

$$sh(f) = \overline{g} \cdot sh(h) + \overline{h} \cdot sh(g) + sh(g) \cdot sh(h). \tag{7}$$

Also, rewriting the equation in a similar manner as AND-2

$$sh(f) = (\overline{g} + sh(g)) \cdot (\overline{h} + sh(h)) - \overline{g} \cdot \overline{h}. \tag{8}$$

Extending this shadow logic function for n-input OR gate (OR-N) yields the general form below

$$sh(f = f_1 + f_2 + \cdots + f_n) = \prod_{i=1}^{n} (\overline{f_i} + sh(f_i)) - \overline{f}. \tag{9}$$

Again, by Lemma 2.1, the shadow logic function for NOR-N is identical to OR-N, namely (9).

3) *Representing XOR Expressions:* The general form of a logic XOR expression is $f = g \oplus h$, where g and h can be logic variables or functions. It can be rewritten using NOT, AND and OR gates. With (1) and (6), the shadow logic function for two-input XOR (XOR-2) can be derived. It is given by

$$sh(f) = sh(g) + sh(h). \tag{10}$$

Extending this shadow logic function for n-input XOR gate (XOR-N) yields the general form in

$$sh(f = f_1 \oplus f_2 \oplus \cdots \oplus f_n) = \sum_{i=1}^{n} sh(f_i). \tag{11}$$

The shadow logic function for XOR-N can be conceptually understood because a change to any input will invert the output. In other words, the output of XOR-N is untainted only when all inputs are untainted.

Now that the shadow logic functions for logic primitives are formally presented, we can construct shadow logic functions for more complex logic functions consisting of different types of gates using (4), (9), and (11). However, it is difficult to see how the complexity of the shadow logic function grows with the number of inputs through this representation. To better understand the complexity for gate-primitives, the following section provides a method for counting the number of minterms in a shadow logic function for gate primitives with different numbers of inputs.

D. Counting Minterms of Logic Primitives

The number of minterms for AND, OR, NAND, NOR, and XOR gates can be counted quantitatively as outlined below.

1) *Number of Minterms for AND-N:* For AND-N, the number of minterms in the shadow logic function can be calculated using (12), in which C stands for the combination operation

$$minterms_{\text{AND-N}} = 2^{2n} - C_n^n \cdot 2^n - \sum_{i=1}^{n} C_n^i \cdot (2^i - 1) \cdot 2^{n-i}. \tag{12}$$

Equation (12) calculates the number of minterms in a shadow logic function by subtracting the untainted minterms from all possible minterms in the shadow logic truth table. For an n-input logic function, there are a total of 2^{2n} entries in the shadow truth table. That is, n inputs and n bit taint for those inputs. First, the cases in which all logic variables are untainted need to be removed. This number is $C_n^n \cdot 2^n$. Then, the cases with different numbers of untainted variables need to be taken into consideration. To be general, assume i inputs among the n inputs are untainted. So i untainted variables need to be selected from n. This is the first factor in the product. For AND-N, an untainted logically false variable will determine the output as untainted. Specifically, all instances containing a "0" for either of the i untainted inputs need to be removed. That is the second factor in the product. Finally, the other $n - i$ tainted variables need to be accounted for. Since tainted variables can take either logic value the third factor in the product accounts for the remaining combinations, namely 2^{n-i}. With this analysis, the number of minterms can be calculated for a varying number of input AND gates as later discussed in the experiments section.

2) *Number of Minterms for OR-N:* The shadow logic function for OR-N is the dual of the logic function for AND-N, which can be seen from

$$A_1 \cdot A_2 \cdots A_n = \overline{\overline{A_1} + \overline{A_2} + \cdots + \overline{A_n}}. \tag{13}$$

As already shown, function f and $\overline{f}$ share the same shadow logic function. So $A_1, A_2, \cdots, A_n$ in the shadow logic function for AND-N needs to be replaced with $\overline{A_1}, \overline{A_2}, \cdots, \overline{A_n}$ respectively in order to obtain the shadow logic function for OR-N. Due to this property, the shadow logic functions for an AND-N gate and an OR-N gate have the same number of minterms. This can be taken a step further; any two functions f and g satisfying the following equation:

$$f(A_1, A_2, \cdots, A_n) = \overline{g(\overline{A_1}, \overline{A_2}, \cdots, \overline{A_n})} \tag{14}$$

have the same number of minterms in their shadow logic functions.

3) *Number of Minterms for* NAND*-N and* NOR*-N:* As stated in Lemma 2.1, invertors change logic values while keeping the taint status unaltered. So any two functions satisfying the following equation have exactly the same shadow logic function:

$$f(A_1, A_2, \cdots, A_n) = \overline{g(A_1, A_2, \cdots, A_n)}. \tag{15}$$

With this property, it can be seen that NAND-N and NOR-N have the same number of minterms as AND-N and OR-N as well.

4) *Number of Minterms for XOR-N:* The shadow logic function for XOR-N is given in (11). To count the number of minterms in the shadow logic function for XOR-N, the untainted entries from the shadowed truth table need to be removed from all possible input combinations. Since all input values need to be untainted to have a taint free output, 2^n possibilities need to be removed from the total number of minterms in a shadow logic function. Equation (16) formalizes this analysis

$$minterms_{\text{XOR-N}} = 2^{2n} - 2^n. \tag{16}$$

Now that we have formally represented shadow logic functions for the gate primitives, these equations can be used to derive shadow logic functions for more complicated circuits. The following section takes this analysis a step further by presenting different methods for generating shadow logic for more complex circuits.

IV. Methods for Shadow Logic Generation

Deriving shadow logic from a logic circuit can be done in two different ways. One we denote as *brute force* and the other *constructive*.

A. The Brute Force Method

The brute force method is based on the definition of flows of information and taint. It works by changing the inputs to a logical function and observing what combinations can cause a difference in the output. For all combinations that cause a change at the output, a minterm is added to the shadow logic function. In this way, the brute force method accounts for only the intended information flows; the shadow logic function generated by it always contains the minimum number of minterms among all shadow logic functions that safely track all the information flows. Thus, by Definition 4, the brute force method is precise for shadow logic function generation.

The brute force method is of high computational complexity because every single input combination needs to be checked in order to accurately determine which minterms are to be added to the shadow logic function. Theorem 3 formally states and proves the complexity of this method.

Theorem 3: The complexity of the brute force method is $O(2^{2n})$.

Proof 3: Using the brute force approach to create the shadow logic function $sh(f)$, the algorithm has to compute the shadow truth table by using the function inputs I and their taint values T as inputs to the shadow truth table. Each row of the shadow truth table requires assignments to the input set I and their taint value set T, and then computing the taint value for the output of that row. For each row, given a function f, its inputs set I, and a subset of inputs T that are tainted, the brute force algorithm checks whether *any* combination of tainted inputs affect the output of f, which is exponential in the number of bits in T. Since this exponential (in T) algorithm has to be executed for *each* row with $\{I\} \cup \{T\}$ inputs, the overall algorithm is exponential in the number of inputs to the function and approaches $O(2^{2n})$. ■

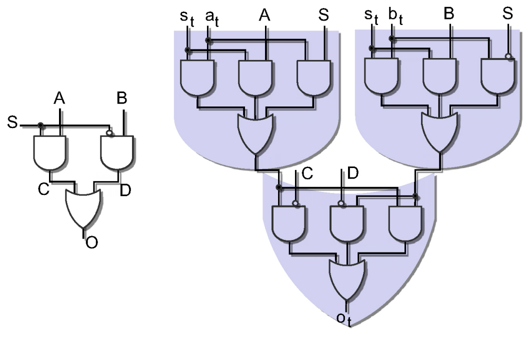
B. The Constructive Method

A less computationally complex alternative to this brute force method is to generate shadow logic for gates in a symbolic manner. This symbolic approach divides the logic expression into logic primitives and generates shadow logic for these subsections constructively in a similar manner to technology mapping. In other words, a library can be built for each gate that creates its corresponding tracking logic. This can be easily done by referring to (4), (9), and (11). The constructive method takes a less complex approach in that it focuses on generating tracking logic for gates of a logic function discretely. The computation complexity of this method is *linear* to the number of gates in a design.

Fig. 3 illustrates how the constructive method generates shadow logic for a two-input multiplexer (MUX-2). First, the logic equation of MUX-2 is translated to a network of NOT, AND, and OR gates. Then, these logic primitives are replaced by their corresponding shadow logic. Finally, proper connections are made to complete the shadow logic circuit.

However, generating shadow logic in this manner is not always guaranteed to be precise. Meaning, a shadow logic function generated constructively can indicate a flow of tainted information propagated from the input to the output when it in fact tainted information does not flow. Such impreciseness results from excessive minterms in the shadow logic function by Definition 4.

As an example, Table II shows the number of minterms in the shadow logic functions for a 4-bit adder generated



(a) (b)

Fig. 3. (a) 2-to-1 multiplexer. (b) Shadow logic of 2-to-1 multiplexer generated using the constructive method.

TABLE II

MINTERM COUNTS OF SHADOW LOGIC FUNCTIONS OF A 4-BIT ADDER GENERATED BY THE BRUTE FORCE AND CONSTRUCTIVE METHODS

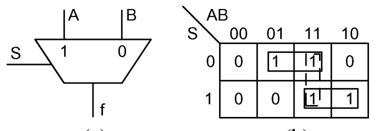
Method	sum[0]	sum[1]	sum[2]	sum[3]	cout
Brute force	229 376	241 664	246 272	248 000	208 160
Constructive	229 376	245 760	251 648	250 656	227 864

by the two different methods discussed. We can see that the number of minterms for the brute force method is less than or equal to that for the constructive method. This means that the constructive method more frequently indicates that information flowed from the input to the output of the logic function. Since the brute force method generates precise shadow logic, the constructive method is actually overly conservative because it contains more minterms.

Such additional minterms are false positives that indicate that a flow of information has occurred when in fact it has not. Taint can quickly propagate throughout the system, e.g., a tainted state machine can taint the whole design in just a few clock cycles or a tainted program counter will quickly cause every bit of information in the processor to become tainted. When a conservative shadow logic function is used for taint propagation, the entire system can get into a tainted state when in fact it is not tainted. At this point, a declassification such as what is presented in [8], from a separation kernel with the highest security level is required to recover the system to a usable state. Generally speaking, being conservative is safe but frequent declassification will make a system unusable. Section V will discuss the imprecision problem of the constructive method in more detail.

V. IMPRECISENESS OF THE CONSTRUCTIVE METHOD

In the previous section, we introduced two existing methods for shadow logic generation. However, the brute force method is computationally expensive for large circuits. Shadow logic circuits generated by the constructive method tend to contain false positives that indicate unintended information flows. This section focuses on the impreciseness of the constructive method. The imprecision is first observed from a simple example. Then, the cause of the impreciseness is stated and analyzed using switching circuit theories. Finally, a solution to this imprecision problem is proposed and formally proved.



(a) (b)

Fig. 4. (a) 2-input multiplexer. The initial function $f = SA + \overline{S}B$, when shadowed constructively, is not precise. (b) Karnaugh map of a 2-input multiplexer. The dotted box indicates the additional term AB that must be added to the original logic function to insure its constructively derived shadow function is precise.

A. *Constructive Method Overly Conservative*

The constructive method in general produces more minterms than the brute force method making it overly conservative. This overly conservative outcome can be seen in the constructive shadowing of a two input multiplexer (MUX-2), whose logic function is $f = SA + \overline{S}B$. Here S is the select line and the inputs are A and B as shown in Fig. 4(a). To shadow MUX-2 constructively, all of the gates in the system need to have shadow logic created for them independently. This includes shadowing both AND gates (SA and $\overline{S}B$) and the OR gate that combines the AND terms. According to the shadow function for OR as shown in (9), the function in (17) needs to be evaluated. Once (17) is expanded using (4) and simplified, the resulting shadow logic function can be seen in (18)

$$sh(f) = \overline{SA} \cdot sh(\overline{S}B) + \overline{\overline{S}B} \cdot sh(SA) + sh(SA) \cdot sh(\overline{S}B) \quad (17)$$

$$sh(f) = \overline{S}b_t + Sa_t + \overline{A}Bs_t + A\overline{B}s_t + ABs_t + a_ts_t + b_ts_t. \quad (18)$$

If the shadow logic function for MUX-2 is computed using the brute force method, (19) will be obtained. There is an extra term ABs_t in the constructive shadow logic function (18) that is not present in the shadow function derived from the brute force method. This term introduces extra minterms in the shadow logic function and makes it overly conservative

$$sh(f) = \overline{S}b_t + Sa_t + \overline{A}Bs_t + A\overline{B}s_t + a_ts_t + b_ts_t. \quad (19)$$

To better understand this, consider the case when S is tainted with A and B are both untainted 1. Using the Karnaugh map in Fig. 4(b), it can be seen that changing S has no affect on the output of the original logic function and it remains logic 1 because $A = B$. However, (18) indicates that the output should be tagged as tainted due to the term ABs_t. The shadow logic function generated by the brute force method does not indicate that the output should be tagged in this case because there is no such term in the shadow logic function. The following section describes the cause for this extra term in the constructive method's shadow logic function.

B. *Cause of Overly Conservative Results*

As shown by (17), there are two steps in the shadow logic function generation process of MUX-2. Shadow logic for terms of the AND gates and OR gate are computed separately to get the final shadow logic function. While generating shadow logic for the term SA, an assumption is being implicitly made that forces all other minterms to appear as false. In other words, all other cases in which $S \neq 1$ and $A \neq 1$ the term is logic false. Constructively shadowing $\overline{S}B$ makes a similar assumption. However, such assumptions cannot be satisfied. Transitioning between two minterms, ABS and $AB\overline{S}$, does not change the output of the original logic function. This assumption forces these switching cases to be handled conservatively by the shadow logic function.

Researchers in the testing area have formalized an abundance of theoretical results in fault effect propagation [23], [24], which is similar to taint propagation in nature. As observed by researchers in the switching circuit area, reconvergent fanout which results in correlation of inputs to the reconvergence gate [25] and logic hazard [26] which causes a spurious output pulse during input change both provide a good insight to the impreciseness of the constructive method. In the MUX-2 example, the shared variable S causes data correlation, which violates the implicit assumptions of the propagation expressions in (9) and thus leads to impreciseness. From another point of view, S and its complement cause a one variable switch logic hazard, which cannot be precisely accounted for by the constructive method.

C. The Complete Sum Approach

As mentioned, when the MUX-2 is shadowed constructively, the resulting shadow logic is overly conservative. This section will prove how to obtain a precise shadow logic function. Before the proof is presented, first observe a solution to the MUX-2 example.

When MUX-2 is shadowed constructively, the problem occurs with the minterms ABS and $AB\overline{S}$ of the original logic function as shown in the dotted box of Fig. 4(b). The simplified term from these two minterms is AB. The Karnaugh map in Fig. 4(b) shows the resulting function with this extra term included. As can be seen, all prime implicants are now included in the function, although AB is non-essential to cover the logic function. By constructively shadowing the original logic function of $SA+\overline{S}B+AB$ the switching case that resulted in this imprecision is no longer present because it is contained in AB. When this function is shadowed constructively, the result is identical to that of the brute force method containing the minimum number of minterms and thus precise.

It should be noted that these switching cases create logic hazards in the function as shown by Eichelberger [26]. Such transitions have a chance of momentarily showing logic false at the output of a logic circuit that has delays present. These single variable hazards are known as static 1 logic hazard. Eichelberger [26] defined logic 1 hazards as a transition from a logic 1 term to another logic 1 term with the output momentarily showing a logic 0 between transitions. Now that an understanding of the cause is has been presented, a solution to the impreciseness of the constructive method can be formalized through a theoretical proof.

Theorem 4: Covering all 1 to 1 single variable switching cases will result in a shadow logic function that is precise when shadowed constructively.

Proof 4: Consider two minterms $m_1 = (A_1, A_2 \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_n)$ and $m_2 = (A_1, A_2 \ldots, A_{i-1}, \overline{A_i}, A_{i+1}, \ldots, A_n)$ of an n input function f, which differ by a single variable switch. When f is shadowed constructively, the shadow logic function will be overly conservative because it has no information about switching between these two minterms. The shadow logic function labels such transitions as a potential information flow. Thus in order to have a precise shadow logic function, all 1 to 1 transitions from single variable switches need to be covered by a term in the original logic function.

Assume the two minterms m_1 and m_2 are covered by an implicant $p = (A_1, A_2 \ldots, A_{i-1}, A_{i+1}, \ldots, A_n)$. The taint status of p is constantly 0 during the switch because there is no tainted input. According to (9), the taint status of m_1+m_2+p, denoted by $sh(m_1+m_2+p)$ is

$$sh(m_1+m_2+p) = \overline{m_1+m_2} \cdot sh(p) + \overline{p} \cdot sh(m_1+m_2) + sh(m_1+m_2) \cdot sh(p)$$

where $sh(m_1+m_2)$ and $sh(p)$ are the taint status of m_1+m_2 and p, respectively.

Because both m_1+m_2 and p are logic true during the switching between m_1 and m_2, the taint status of m_1+m_2+p will be dominated by the taint status of p, which is constantly 0. Thus, the false positives caused by the switching can be eliminated. ■

For example, consider the logic function of MUX-2 as shown in Fig. 4(b). Here the transition indicated by the box must be covered because it is a single variable switching case. In doing so the most precise shadow logic function will be obtained for MUX-2 as already discussed.

Theorem 5: A function containing all prime implicants will cover all 1 to 1 single variable switches.

Proof 5: By the definition of logic hazard [26], a function must have all of its 1 to 1 transitions covered to be free of logic hazards. To rid the function of all logic hazards, all prime implicants are needed as proved by Eichelberger [26]. The proof is repeated here for completeness. At least one of the logic terms in a function must remain logic 1 during a transition in order to avoid a static 1 logic hazard. Assume that some prime implicant $(A_i, A_{i+1}, \ldots, A_j)$ is not included in the logic function. The transition from $(A_1, A_2 \ldots, A_{i-1}, A_i, A_{i+1} \ldots, A_j)$ to $(\overline{A_1}, \overline{A_2} \ldots, \overline{A_{i-1}}, A_i, A_{i+1} \ldots A_j)$ only has a single term $(A_i, A_{i+1}, \ldots, A_j)$ which is sure to remain 1 during the entire transition. Thus, this transition contains a logic hazard and all prime implicants are needed to rid the function of all logic hazards. ■

Theorem 6: A function containing all prime implicants will generate the most precise shadow logic function when shadowed constructively.

Proof 6: All 1 to 1 transitions must be covered in order for a function to produce a precise shadow logic function when shadowed constructively. Theorem 5 states that a function with all prime implicants will cover all 1 to 1 transitions. Thus, a function with all prime implicants will have precise shadow logic by Theorems 4 and 5. ■

Lemma 6.1: The shadow logic function for a function containing static 1 logic hazards will accurately detect the potential flow of information from the input to the output caused by this momentary logic 0 pulse.

TABLE III
PERCENTAGE OF EXCESSIVE MINTERMS FOR SHADOW LOGIC FUNCTIONS AFTER SEVERAL EXPANSION STEPS

Steps	f_1	f_2	f_3	f_4
1	3.12%	7.81%	12.89%	17.68%
2	0%	4.68%	9.96%	14.94%
3		2.34%	7.62%	12.74%
4		0%	5.27%	10.55%
5			3.52%	8.79%
6			1.76%	7.03%
7			0%	5.27%
8				3.96%
9				2.64%
10				1.32%
11				0%
Brute force	44	176	632	2168

The last row shows the number minterms in a precise shadow logic function.

As a sanity check, consider some simple logic expressions to see how the number of minterms decreases as the function is expanded. Consider the following four unrelated functions:

$$\begin{aligned} f_1 &= AB + \overline{B}C \\ f_2 &= AB + \overline{B}C + \overline{C}D \\ f_3 &= AB + \overline{B}C + \overline{C}D + \overline{D}E \\ f_4 &= AB + \overline{B}C + \overline{C}D + \overline{D}E + \overline{E}F. \end{aligned}$$

Table III lists the percentage of excessive minterms, i.e., false positives in a conservative shadow logic function that indicate non-existing information flows, after each expansion step and the number of minterms in the precise shadow logic function generated by the brute force method. Expansion here is the processes of producing another term from the existing terms. For example, $f_1 = AB + \overline{B}C$ expands to $f_1 = AB + \overline{B}C + AC$. This percentage shows the amount of excessive minterms that are not present in the precise shadow logic function. The leftmost column indicates the number of expansion steps taken in the original logic function and the bottom row shows the number of minterms in the precise shadow logic function generated by the brute force method. As shown, with the addition of a prime implicant to the original logic function through each expansion step, the number of minterms of the shadow logic function decreases until the function reaches a complete sum, i.e., having all prime implicants [27]. At this step it is precise because it contains no excessive minterms, which is a re-enforcement of our theoretical proof.

From the theoretical proof and sanity check, we deduce another method to derive precise shadow logic. We call this the *complete sum approach*.

Definition 5—complete sum approach: Constructively shadowing a logic function in the complete sum form, i.e., having all prime implicants, to generate shadow logic.

This approach differs from the conventional constructive method in that it requires the logic function being shadowed to have all its prime implicants included. It will produce an equally precise shadow logic function as the brute force method as proved. However, the maximum number of prime implicants for an n-input function approaches $O(3^n/\sqrt{n})$ [28], [29]. The problem of determining if a product term is a prime implicant of a function is between $NP \cup coNP$ and $\sum_2^P$ in complexity [29]. These make the complete sum approach expensive for large designs.

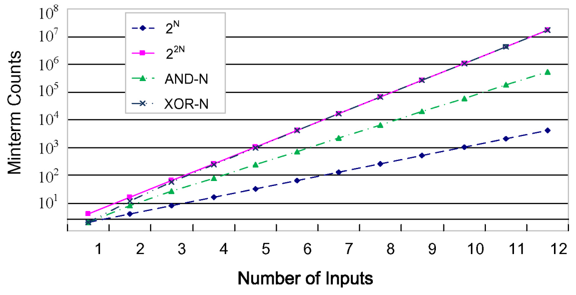


Fig. 5. Shadow logic function of gate primitives grows exponentially on the number of inputs. AND, OR, NAND, and NOR all have the same number of minterms as discussed in Section III. This plot is in log scale.

In previous sections, the fundamental properties of GLIFT were discussed and various shadow logic generation methods were introduced, which reveals how information flows can be tracked precisely at the gate level. An effective solution is provided to overcome the impreciseness of the constructive method as observed from the MUX-2 example. In the next section, we present some quantitative analysis using *ISCAS* and *IWLS* benchmarks to show complexity and other factors such as area, delay and preciseness that should be taken into consideration when employing GLIFT in real applications.

VI. EXPERIMENTAL RESULTS

In order to more concretely understand how the shadow logic scales, this section discusses shadow logic when it is applied to different logical constructs and benchmarks. First, complexity analysis through minterm count is covered in Section VI-A. In this analysis, the number of minterms is used as our complexity metric because it conveys the complexity of the results independent of optimizations. It allows us to accurately show how the complexity of the problem grows with an increasing number of inputs to the original logic function. Subsequently, in Section VI-B, *ISCAS* and *IWLS* benchmarks are shadowed with different shadow logic generation methods to analyze the tradeoff between area, delay and preciseness.

A. Minterm Count Analysis

This section shows experimental results in terms of the number of minterms of a shadow logic function. Quantitative calculations are done on a varying number of input AND, OR, XOR, NAND, and NOR gates using the counting method in Section III-D. Tests are also run on combinational *ISCAS* benchmarks and non-*ISCAS* benchmarks using the brute force method. The results for logic gates can be seen in Fig. 5 in which the number of minterms corresponds to the amount of information that flows from the inputs to the outputs.

As discussed in Section III-D, AND, OR, NOR, and NAND all have the same number of minterms due to Theorem 2 and Lemma 2.1. The results show that as the number of inputs increases, the complexity of the shadow logic function

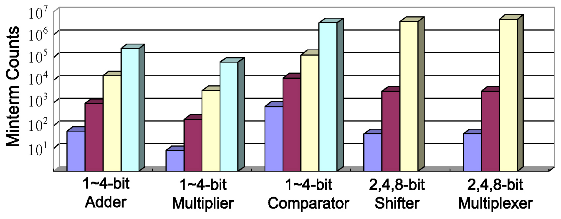


Fig. 6. Shadow logic for more complicated logical functions also grows exponentially on the number of inputs.

increases exponentially. The plot, in log scale, shows the total number of minterms possible for the original logic function 2^n and the shadow logic function 2^{2n}. For XOR, the number of minterms approaches the total number of minterms for a shadow logic function with n inputs. This is due to the property of XOR that requires each input to be untainted in order for the result to be untainted.

The number of minterms for shadow logic functions of some more complex logic circuits can also be realized using a brute force approach. The results are for *ISCAS* benchmarks 74L85 (4-bit magnitude comparator), 74 283 (4-bit adder), and s344/s349 (4×4 add-shift multiplier). Results are also shown for a simple bit shifter and a multiplexer with different numbers of inputs. We were limited to the number of *ISCAS* benchmarks we could test due to the exponential increase in the number of minterms on the number of inputs.

Similar behavior shown for gate primitives also holds for more complicated logic blocks. Graphically these numbers can be represented in Fig. 6. The figure is grouped into sections based on the logic blocks tested. Specifically 1, 2, 3, and 4 bit adders, multipliers, and comparators were tested as well as 2, 4, and 8 bit shifters and multiplexers. Fig. 6 shows that, similar to the gate primitives, the number of minterms for more complex logic blocks also increases exponentially on the number of inputs $O(2^{2n})$.

Minterm count on basic gates and *ISCAS* benchmarks shows that tracking data at the gate-level becomes difficult to manage because of the large increase in the number of minterms. Since the area, delay and preciness for a circuit are heavily dependent on the optimization used, the following section provides insight on the tradeoffs between these three factors when applied to several benchmarks.

B. Area, Delay, and Preciseness Analysis

We have already shown by a theoretical proof and examples that the complete sum approach provides a solution to the impreciseness of the constructive method. However, area and delay are also important factors that should be taken into account. This is especially true for the shadow logic since it can easily dominate the total logic consumption. We carried out experiments on several *ISCAS* and *IWLS* benchmarks to obtain area and delay reports for shadow logic circuits created by the discussed shadow logic generation methods. *ABC* [30] is used as the synthesis tool because it provides various synthesis commands and scripts for different optimization goals. The *resyn2* synthesis script in *ABC* is used to optimize the shadow logic circuits in our experiment because it provides a good tradeoff between area and delay.

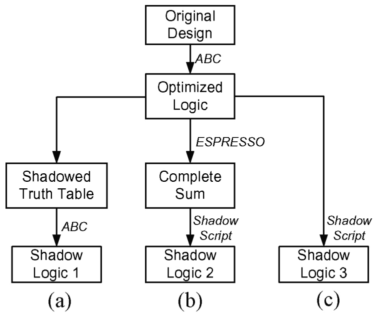


Fig. 7. Design flows of different shadow logic generation methods. (a) Brute force method. (b) Complete sum approach. (c) Constructive method.

In our experiment, shadow logic circuits are generated through three different flows: brute force, complete sum and constructive as shown in Fig. 7. The complete sum flow generates functionally equivalent shadow logic circuits as brute force at a relatively lower computational complexity. It differs from the constructive flow in that the original logic is expanded to complete sum representation before being shadowed. The complete sum flow is listed as a comparison to the other two flows. *ISCAS* and *IWLS* benchmarks are optimized using the *resyn2* synthesis script before being shadowed. Different methods are processed with different design flows as shown in Fig. 7.

1) *The brute force method* works on a full shadow truth table containing both the original logic variables as well as their taints as inputs and outputs. *ABC* takes in the shadow truth table and generates a shadow logic function for the benchmark.
2) *The complete sum approach* uses *ESPRESSO* [31] to generate complete sum representation of the benchmark. Then the complete sum is shadowed constructively using our own shadowing script to produce a shadow logic function.
3) *The constructive method* shadows the optimized logic circuit directly using our own shadowing script to create a shadow logic function.

Finally, all the shadow logic circuits are synthesized with the *resyn2* script again and mapped to the *MCNC* library for area and delay reports.

Table IV shows some statistics of the benchmarks used for area and delay analysis. Among the benchmarks, 74 283 is a 4-bit full adder; *alu4_cl* is a 4-bit arithmetic logic unit; *s344/s349* is 4-bit multiplier; *des* represents the data encryption standard and the rest of the benchmarks are glue logic.

Table V gives the area/delay of shadow logic circuits generated by these different methods. As an example, the original design of *alu4_cl* reports an area/delay of 671/11.9. These area measurements are generated from *ABC* and the units are not provided. The shadow logic for *alu4_cl* generated by the brute force method, complete sum approach and constructive method have areas/delays of 5824/23.9, 2876/22.5, and 2292/21.5, respectively. We can see that the shadow logic is much more complex than the original design since it can easily dominate the total logic consumption.

TABLE IV
STATISTICS OF THE BENCHMARKS USED FOR AREA AND DELAY ANALYSIS

Benchmark	# of Inputs	# of Outputs	# of Gates
74 283	9	5	197
alu4_cl	10	6	308
s344/s349	9	8	319
9symml	9	1	152
C5315	178	123	1011
C7552	207	108	1197
i10	257	224	1540
C6288	32	32	2022
too_large	38	3	3130
des	256	245	3085

TABLE V
AREA/DELAY OF ORIGINAL DESIGN AND CIRCUITS FROM DIFFERENT SHADOW LOGIC GENERATION METHODS

Benchmark	Original	Brute Force	Complete Sum	Constructive
74 283	109/6.2	187/9.2	177/9.6	179/10.8
alu4_cl	671/11.9	5824/23.9	2876/22.5	2292/21.5
s344/s349	488/10	1760/18.4	1761/19.4	1765/19.5
9symml	133/12.8	2549/19.5	2569/19.4	2528/19.4
C5315	2501/25.5	-/-	-/-	9166/36
C7552	2897/21.3	-/-	-/-	9377/30.9
i10	3583/27.9	-/-	-/-	14 407/52.9
C6288	4994/76.4	-/-	-/-	11 965/93.8
too_large	7812/14.5	-/-	-/-	15 957/16.7
des	14 539/16	-/-	-/-	75 262/23.7

The brute force and complete sum methods are functionally equivalent. We have formally proven this and did a sanity check using formal equivalence checking using the *cec* command from the *ABC* tool. Theoretically, an optimal logic synthesis tool would generate the same optimal logic circuit for both methods. However, since the two shadow logic functions are initially described differently, they lead to different implementation results. Logic optimization is a hard problem and the representation of the shadow logic function can cause variations in implementation results. We have found that this is the case with a variety of logic synthesis tools including *Xilinx ISE, Altera Quartus II*, and *Synopsys Design Compiler*.

We were limited to the designs that we could test because the brute force method took too long to test on functions with a large number of inputs (over 12 bits of inputs) due to the complexity of the algorithm. In addition, finding all prime implicants for a large function is a hard problem. The maximum number of prime implicants for an n-input function approaches $O(3^n/\sqrt{n})$ [28], [29]. The problem of determining if a product term is a prime implicant of a function is between $NP \cup coNP$ and $\sum_2^P$ in complexity [29]. As a result, designs with a relatively small number of inputs can have their shadow logic generated using the brute force method and complete sum approach but larger ones cannot. For this reason, *des* and the other large benchmarks have no area and delay reports from these two methods.

The complexity of these shadow logic generation methods can be seen from their execution time, which is shown in Table VI. The brute force method takes the longest time to complete because its complexity is $O(2^{2n})$. The constructive method reports the shortest execution time, which is linear to the number of gates in a design. One needs to find all prime implicants of an original logic function in the complete sum approach. This process is time-consuming for large designs [28], [29]. The tool [31] we used was unable to handle large bench marks. That's why the brute force and complete sum methods do not have runtime results for large designs.

TABLE VI
RUNTIME REQUIRED BY DIFFERENT METHODS FOR SHADOW LOGIC GENERATION

Benchmark	Brute Force (hh:mm:ss.cs)	Complete Sum (ss.cs)	Constructive (ss.cs)
74 283	0:08:39.26	0.45	0.07
alu4_cl	1:53:23.74	0.43	0.23
s344/s349	0:01:28.51	0.48	0.24
9symml	0:09:27.44	1.12	0.14
C5315	-/-	-/-	1.01
C7552	-/-	-/-	1.14
i10	-/-	-/-	1.36
C6288	-/-	-/-	1.33
too_large	-/-	-/-	2.41
des	-/-	-/-	2.56

As discussed in previous sections, the constructive method can be conservative when some prime implicants are not included; the complete sum approach fixes this impreciseness by adding prime implicants. By counting the number of minterms in the shadow logic function generated with the brute force, complete sum and constructive methods, a better intuition of the impreciseness of the constructive method can be established.

Fig. 8 gives the percent of minterms with respect to the total number of minterms, which is 2^{2n} for an n input original logic function. The impreciseness of the constructive method can be seen from larger minterm counts, which are listed bottommost. As an example, for the *cout* bit of 74 283, shadow logic generated by the brute force method and the complete sum approach both tag 79.4% of the total input combinations as tainted while the shadow logic created by the constructive method tags to 86.9%. It is also worthy noticing that some outputs are tainted up to 90% of the total input combinations. For these outputs, it may be more beneficial to use a simple logic OR of all the related taint inputs instead of complex shadow logic because area and delay can be greatly reduced. Although this will result in impreciseness it can be tolerated for reduced area and delay.

As shown by experimental results, the shadow logic circuits generated by the brute force method and complete sum approach are both precise while those created by the constructive method can be conservative. However, shadow logic functions created by the brute force method and complete sum approach may still report different area and delay since they are initially described differently. Furthermore, the brute force method and the complete sum approach are both of high complexity, which makes them difficult to compute for large designs. Tradeoffs among area, delay, preciseness and computational complexity

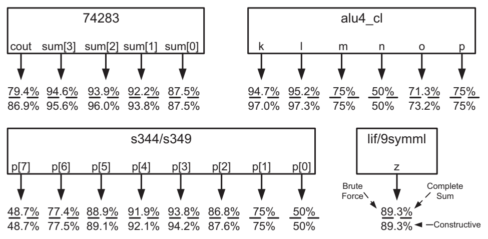


Fig. 8. Percentage of tainted minterms in shadow logic circuits generated by the brute force, complete sum, and constructive methods. As we have proved shadow logic functions generated by the brute force and complete sum methods contain the same number of minterms since they are both precise.

is to be taken into account in shadow logic circuit design. Although precision difference of shadow logic functions generated by different methods can be small but may still be important for highly secure applications, e.g., in our secure processor [7], we needed to precisely create shadow logic for MUXs because they are an essential building block of our design. If these are not handled precisely, the entire system can quickly become tainted making any reasonable conclusion about the information flows nearly impossible.

VII. Conclusion

This paper presented the theory behind the shadow logic for systems that implement GLIFT. It provides a more concrete understanding to gate level information flow tracking by formally presenting terms and definitions, proving essential properties, formalizing the shadow logic functions for logic primitives, and introducing a symbolic approach for tracking logic generation. It also presents the imprecision problem with the constructive method and presents a proof to show how the most precise shadow logic function can be obtained without using the computationally complex brute force approach.

Experimental results have shown that the number of minterms in a shadow logic function grows exponentially with the number of inputs to $O(2^{2n})$. In highly secure applications where high precision is required, sacrifices in terms of the complexity of generating the precise shadow logic will need to be made. For example, if non-interference is to be guaranteed between the flight control and user network on an aircraft, overly conservative approaches will likely generate too many false-positives in order to make a reasonable conclusion about what caused the policy violating information flows. More precision will reduce the false-positives and build more confidence in what is causing the unintended information flows. The amount of precision that is required to adequately meet the information flow policy of an application is an open problem and reserved for future research.

Area and delay overheads are critical issues if the shadow logic is to be deployed in an application for dynamic information flow tracking. However, if GLIFT were to be deployed dynamically, only high integrity regions of the system should be monitored in case the overheads could not be tolerated. The designer could choose what critical regions should be monitored to reduce this huge overhead. Additionally, optimized shadow logic for larger components such as *multiplexer*, *comparator* and *alu* can be built as macros to reduce area and delay overhead. Moreover, GLIFT can also be added to a design and test *statically* if a design complies with pre-defined information flow policies. Our future work will concentrate on optimized shadow logic circuit generation which will enable the wide application of GLIFT as a security enhancement.

Acknowledgment

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

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. Security Privacy*, Apr. 1982, pp. 11–20.
[2] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Bedford, MA, Tech. Rep. MTR-2547, vol. 1, 1973.
[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, pp. 5–19, Jan. 2003.
[4] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *Proc. 20th SOSP*, Oct. 2005, pp. 17–30.
[5] 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. ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6. 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 Int. Conf. Architectural Support Programming Languages Operating Syst.*, 2004, pp. 85–96.
[7] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proc. 14th Int. Conf. Architectural Support Programming Languages Operating Syst.*, 2009, pp. 109–120.
[8] 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. Microarchitecture (Micro)*, Dec. 2009, pp. 493–504.
[9] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Theoretical analysis of gate level information flow tracking," in *Proc. DAC*, 2010, pp. 244–247.
[10] S. Zdancewic, "Challenges for information-flow security," in *Proc. 1st Int. Workshop PLID*, Aug. 2004.
[11] D. E. Denning, *Cryptography and Data Security*. Reading, MA: Addison-Wesley, 1982.
[12] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *J. Comput. Security*, vol. 4, nos. 2–3, pp. 167–187, 1996.
[13] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Trans. Programming Languages Syst.*, vol. 25, no. 1, pp. 117–158, Jan. 2003.
[14] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. (2001, Jul.). *Jif: Java Information Flow (Software Release)* [Online]. Available: http://www.cs.cornell.edu/jif
[15] A. Hurst. (2004, Jun.). *Analysis of Perl's Taint Mode* [Online]. Available: http://hurstdog.org/papers/hurst04taint.pdf
[16] R. Gupta, N. Gupta, X. Zhang, D. Jeffrey, V. Nagarajan, S. Tallam, and T. Chen, "Scalable dynamic information flow tracking and its applications," in *Proc. NGS Workshop*, Apr. 2008, pp. 1–5.
[17] C. Lewis and C. Sturton. (2008, May). SHIFT+M: Software-hardware information flow tracking on multi-core. Dept. Elec. Eng. Comput. Sci., Univ. California Berkeley, Berkeley, Res. Project [Online]. Available: www.eecs.berkeley.edu/∼csturton/classes/cs258/ShiftM_Final.pdf
[18] 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.
[19] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A programmable accelerator for dynamic taint propagation," in *Proc. 14th IEEE Int. Symp. HPCA*, Feb. 2008, pp. 173–184.

[20] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *Proc. 35th ISCA*, Jun. 2008, pp. 377–388.

[21] D. J. Bernstein. (2005, Apr.). Cache-timing attacks on AES. Dept. Math., Statist., Comput. Sci., Univ. Illinois Chicago, Chicago, Tech. Rep. cd9faae9bd5308c440df50fc26a517b4 [Online]. Available: http://cr.yp.to/papers.html#cachetiming

[22] Federal Aviation Administration. *Boeing Model 787-8 Airplane; Systems and Data Networks Securityisolation or Protection From Unauthorized Passenger Domain Systems Access* [Online]. Available: http://cryptome.info/faa010208.htm

[23] F. C. Wang, *Digital Circuit Testing: A Guide to DFT and Other Techniques*. New York: Academic, Aug. 1991.

[24] M. H. Schulz and D. Pellkofer, "A three-valued fast fault simulator for scan-based VLSI-logic," in *Proc. 1st Eur. Test Conf.*, Apr. 1989, pp. 41–48.

[25] F. Maamari and J. Rajski, "A method of fault simulation based on stem regions," *IEEE Trans. Comput.-Aided Des.*, vol. 9, no. 2, pp. 212–220, Feb. 1990.

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

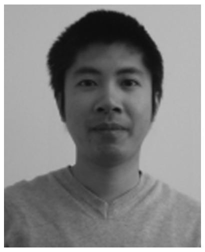
[27] E. J. McCluskey, *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill, 1965.

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

[29] H. S. Nguyen, "Approximate Boolean reasoning: Foundations and applications in data mining," *Trans. Rough Sets V*, vol. 4100, pp. 334–506, 2006.

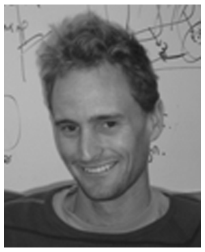
[30] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*, Release 70930 [Online]. Available: http://www.eecs.berkeley.edu/∼alanmi/abc

[31] Berkeley Donald O. Pederson Center for Electronic Systems Design. *Espresso: A Multi-Valued PLA Minimization*, Ver. 2.3 [Online]. Available: http://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm



Wei Hu is currently pursuing the Ph.D. degree from the School of Automation, Northwestern Polytechnical University, Xi'an, Shaanxi, China.

He is a Visiting Graduate Student with the Department of Computer Science and Engineering, University of California, San Diego, from September 2010 to September 2011. His current research interests include 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 Post-Doctoral Scholar 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 B.Tech. degree in computer science from the Indian Institute of Technology Guwahati, Guwahati, India, in 2005. He is currently pursuing the Ph.D. degree from the Department of Computer Science, University of California, Santa Barbara.

His current research interests include computer architecture and run-time analyses to program security and debugging.

Mr. Tiwari received the Best Paper Award at Parallel Architectures and Compilation Techniques in 2009 and the IEEE Micro Top Pick in 2009.

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 IEEE 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. 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.