Theoretical Analysis of Gate Level Information Flow Tracking

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

ABSTRACT

Understanding the flow of information is an important aspect in computer security. There has been a recent move towards tracking information in hardware and understanding the flow of individual bits through Boolean functions. Such gate level information flow tracking (GLIFT) provides a precise understanding of all flows of information and is an invaluable tool for proving many system security properties. This paper presents a theoretical analysis of GLIFT. It formalizes the problem, provides fundamental definitions and properties, introduces precise symbolic representations of the GLIFT logic for basic Boolean functions, and gives analytic and quantitative analysis of the GLIFT logic.

1. INTRODUCTION

Many computing systems are often entrusted with confidential data, and therefore the need to ensure that these systems are secure is crucial. Determining the exact flow of information throughout a system is paramount in verifying that data is secure. Information flow tracking (IFT) is a tool that can be used to identify security vulnerabilities, so that they can be handled in an appropriate manner. IFT associates a tag with a piece of data; the tag indicates a specific property about that information.

Initial research in this domain concentrated on software mechanisms to monitor information flow [1][2][3]. Although effective in some applications, the underlying hardware is capable of leaking information through timing, power, thermal and other covert channels. These information flows can only be exposed with an understanding of the hardware upon which the software is running. Most existing hardware techniques for IFT tag information are at the instruction or register level [4][5]. This leads to overly conservative or imprecise IFT because entire words are being flagged as tainted rather than specific bits. In order to have more precise IFT, data should be tracked at the gate level. This is known as Gate Level Information Flow Tracking (GLIFT).

GLIFT, as proposed by Tiwari et al. [6], is a form of IFT that tracks bits precisely at the gate level. They define a method to generate shadow logic that monitors the flow of information through Boolean gates. This is a more precise method for tracking information, and provides a deeper understanding of information flow through the computing system. They use GLIFT to design a more secure computing system, e.g., a processor that ensures strict non-interference [7].

DAC'10, June 13-18, 2010, Anaheim, California, USA

This paper expands on previous work by providing a theoretical foundation for generating and understanding Shadow (or GLIFT) Logic as used by [6][7].

The major contributions of this paper are:

1) **Defining and proving fundamental properties of GLIFT:** We precisely define the idea of *taint* and prove fundamental properties of the GLIFT logic.
2) **Providing a precise symbolic representation of the GLIFT logic for basic Boolean functions:** We derive a symbolic representation of the shadow logic and formally analyze the properties of common logical constructs including gate primitives and multiplexers.
3) **Analytic and quantitative analysis of GLIFT logic:** We present and analyze properties of the GLIFT logic for basic Boolean functions and a number of ISCAS benchmarks.

The remainder of this paper is organized as follows. Section II provides details about shadow logic generation including a symbolic representation of the Shadow Logic Function. Section III we provide experimental results of our shadow logic implementations using ISCAS benchmarks. We conclude in Section VI.

2. THEORY

Shadow logic is the co-existing logic that propagates information throughout a system. This logic performs no operations that affect the way that the system operates. The shadow logic is used to understand where information is propagating to point out where potential security vulnerabilities are. Before the methods for generating shadow logic can be described, the definition of *taint* needs to be understood. Throughout this discussion, the shadow logic for AND/OR gates and a 2-input MUX will be discussed.

Definition 1—Taint: Taint is a tagged associated with a bit that indicates that this information should be tracked. If an output is dependent on the value of a tainted input, then the taint is said to propagate from the input to the output.

Thus if toggling the logical value of a particular tainted input changes the output, a term needs to be added to the shadow logic to cover this specific case. To generalize the definition of taint, the next subsection describes taint in much more detail. For example, a 2-input AND gate where one of the inputs is un-tainted with the value of '0' means that the output is always un-tainted regardless of the other input value and its taint.

2.1 Fundamental Properties of GLIFT

Upper-cased letters with a subscript, $A_1, A_2, \cdots, A_n$, are used to denote logic variables and lower-cased letters with a subscript, $a_1, a_2, \cdots, a_n$, are used to denote the taint of $A_1, A_2, \cdots, A_n$ respectively. f, g and h are used to denote logic functions and $sh(f)$, $sh(g)$ and $sh(h)$ to denote the Shadow Logic Functions for f, g and h respectively. When the taint of a logic variable is '0', the logic variable is not tainted. On the other hand, if the taint of a logic variable is '1', this variable is tainted. Throughout this discussion, shadow and taint will be used interchangeably.

Definition 3—Taint of Original Logic Function: The Original Logic Function is said to be tainted when toggling tainted input(s) of the function leads to a change at the output. In other words, a flow of information from a tainted input propagates to the output.

Definition 4—Shadow Logic Function: The Shadow Logic Function is a logic function of the Original Logic Function variables, their inverses and their shadow values, given in the form of

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

The Shadow Logic Function will output logic true whenever the Original Logic Function gets into a tainted state and logic false when there is no information flow through the Original Logic Function. The Shadow Logic Function is the sum of minterms composed of the logic variables, their inverses and their shadow values.

Theorem 1: A simplified Shadow Logic Function will contain only one of either A_i, $\overline{A_i}$, or a_i.

Proof: Only tainted logic variables have a chance to taint the Original Logic Function. Thus a simplified Shadow Logic Function contains only the shadow values of the logic variables but not their inverses. Meaning, terms containing $\overline{a_i}$ $(i = 1,2,\cdots,n)$ are not present in a simplified Shadow Logic Function.

If the product of A_i and a_i $(i = 1,2,\cdots,n)$ appears in the 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:

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

By the definition of taint, changing a tainted input needs to be tracked by the Shadow Logic Function, thus another term:

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

will also taint the output since the logic values and shadow values of the rest of the logic variables are kept unchanged. Thus terms with products of A_i and a_i always appears in pairs (e.g. $m + m'$) in the Shadow Logic Function. This leads sub-terms m and m' to be automatically reduced.

Lemma 1.1: The Shadow Logic Function of a single variable function is the taint of that logic variable. This is because the logic variable and its taint cannot appear at the same time in the Shadow Logic Function.

Theorem 2: The inverse of a Shadow Logic Function is equal to the Shadow Logic Function itself.

Proof: As mentioned, all changes to tainted inputs of a function need to be tracked in order to capture all flows of information. So single variable logic functions such as $g = A$ and $g' = \overline{A}$ share the same Shadow Logic Function $sh(g) = sh(g')$.

Lemma 2.1: Inverters change the value of the logic input but do not affect the taint bit. In other words, the taint bits are propagated from input to output.

Lemma 2.2: Inverting the Original Logic Function does not invert the Shadow Logic Function.

Now that taint is defined, the next subsections discuss the different ways in which Shadow Logic can be derived. These methods include a Brute Force and Symbolic approach.

2.2 Deriving a Shadow Logic Function

One approach is a Brute Force one that is based on the definition of flows of information and taint. The method works by changing the inputs to a logical function and observing what combinations cause a difference in the output. If a change occurs, minterms are added to the Shadow Logic Function for the input combinations that caused this change.

An alternative to this Brute Force method is to use a generalized form for obtaining Shadow Logic for AND/OR gates in a symbolic manner. This S*ymbolic* AND/OR approach divides the problem into subsections and generates shadow logic for these sections. To get precise results, this approach requires some assumptions. Specifically, all logic AND expressions are assumed to be simplified and all sub terms in logic OR expressions do not contain the same logic variable. Stated more formally:

1. If a logic AND expression f is the product of two sub logic AND terms, g and h, then g and h should not contain the same logic variable. Otherwise f is not simplified or constantly logic zero. Cases such as:

$$f = a \cdot ab \quad g = ab \cdot \bar{b}$$

violate this assumption.

2. If a logic OR expression f is the sum of two sub logic AND terms, g and h, then g and h should not contain the same logic variable. Cases such as:

$$f = a + ab \quad g = ab + c\bar{b}$$

violate this assumption.

Representing AND expressions: The general form of a logic AND expression is $f = g \cdot h$. Using the logic function for AND_2 (2-input AND), the Shadow Logic Function for inputs g and h can be realized. We denote $\mathrm{sh}(f), sh(g)$ and $sh(h)$ as the shadow logic function of f, g and h respectively. The resulting shadow logic is shown below in Equation (1):

$$sh(f) = g \cdot sh(h) + h \cdot sh(g) + sh(g) \cdot sh(h) \qquad (1)$$

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

$$sh(f) = \big(g + sh(g)\big) \cdot \big(h + sh(h)\big) - g \cdot h \qquad (2)$$

Although Equation (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., un-tainted true or tainted. The minus sign here means removing the term from the expression. It is important to note that $g \cdot h = 1$ should not affect the output of the Shadow Logic Function and thus needs to be removed.

The same methodology can be used for AND_3. Assume $f = g \cdot h \cdot k$. The same constraints apply to the inputs of AND_3 here. Using the above formula, the result below is obtained:

$$sh(f) = \big(g + sh(g)\big) \cdot \big(h + sh(h)\big) \cdot \big(k + sh(k)\big) - g \cdot h \cdot k$$

With the minus operation, which means removing the term that follows it from the logic function, a general form of the AND_N Shadow Logic Function can be given as Equation (3) (multiply is defined for logic function with meaning of logic AND):

$$sh(f = f_1 \cdot f_2, \cdots, f_n) = \prod_{i=1}^{N} \big(f_i + sh(f_i)\big) - f \qquad (3)$$

Additionally, the Shadow Logic Function for $NAND_N$ is identical to Equation (3) by Theorem 2.

Representing OR expressions: The general form of a logic OR expression is $f = g + h$ where g and h are independent, i.e. do not contain the same logic variables. We use sh(f), $sh(g)$ and $sh(h)$ to denote the Shadow Logic Functions of f, g and h respectively. As shown by Theorem 2, $sh(f) = sh(\bar{f})$ is true for any logic function. So the result for the AND_2 expression can be used to directly obtain the shadow logic for OR_2. Rewriting the Logic Function for OR_2 using De Morgan's Law the following holds:

$$f = \overline{\bar{g} \cdot \bar{h}}$$

Thus, Theorem 2 states that $\bar{f} = \bar{g} \cdot \bar{h}$ has the same shadow logic for a function $f = \bar{g} \cdot \bar{h}$ and the same approach can be used for OR_2 as it was for AND_2.

$$sh(f) = sh(\bar{f}) = \bar{g} \cdot sh\big(\bar{h}\big) + \bar{h} \cdot sh(\bar{g}) + sh(\bar{g}) \cdot sh(\bar{h})$$

Because $sh(\bar{g}) = sh(g)$ and $sh\big(\bar{h}\big) = sh(h)$, the result is:

$$sh(f) = \bar{g} \cdot sh(h) + \bar{h} \cdot sh(g) + sh(g) \cdot sh(h)$$

Also, rewriting the equation in a similar manner as AND:

$$sh(f) = \big(\bar{g} + sh(g)\big) \cdot \big(\bar{h} + sh(h)\big) - \bar{g} \cdot \bar{h}$$

Extending this Shadow Logic Function for OR_N yields the general form below (4):

$$sh(f = f_1 + f_2 + \cdots + f_n) = \prod_{i=1}^{N} \big(\bar{f}_i + sh(f_i)\big) - \bar{f} \qquad (4)$$

Again, by Theorem 2, the Shadow Logic Function for NOR_N is identical to OR_N, namely Equation (4).

Representing XOR expressions:

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 un-tainted only when all inputs are un-tainted. If lower-cased letters with a sub-script are used to denote the taint bit for the logic variables, the Shadow Logic Function for XOR_N will be as follows:

$$sh(f) = a_1 + a_2 + \cdots + a_n$$

Representing a 2-to-1 MUX: The logic form of a 2-input Multiplexer requires a different approach than AND or OR because the terms of the logic expression are not independent.

The logic function of MUX_2 is $f = g \cdot s + h \cdot \bar{s}$. Denote $sh(f)$, $sh(g)$, $sh(h)$ and $sh(s)$ as the Shadow Logic Functions for f, g, h and s respectively. The logic function of MUX_2 contains dependent terms ($gs \cdot h\bar{s} = 0$), so the formula for OR_2 cannot be used to create a precise Shadow Logic Function. Instead, it needs to be treated as another basic unit and have its Shadow Logic Function analyzed separately.

Equation (5) shows the Shadow Logic Function for MUX_2. The first term shows that when s is un-tainted true, g determines if f is tainted. The second term shows when s is un-tainted false, h determines if f is tainted. The third term shows that when s is tainted, the only case that should produce an un-tainted output is when g and h are both un-tainted and g equals to h. Thus if $g \neq h$ and s is tainted, then the result will be tainted. From the analysis, we can get the precise Shadow Logic Function for MUX_2 if g, h and s are independent.

$$sh(f) = s \cdot sh(g) + \bar{s} \cdot sh(h) + \big((g \oplus h) + sh(g) + sh(h)\big) \cdot sh(s) \qquad (5)$$

It is difficult to see how the complexity of the Shadow Logic Function grows with the number of inputs to a logic function with 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.

2.3 Counting Minterms of Logic Primitives

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

Number of minterms for AND_N: For AND_N, the number of minterms in the Shadow Logic Function can be calculated using Equation (6).

$$num = 2^{2N} - C_N^N \cdot 2^N - \sum_{i=1}^{N-1} C_N^i \cdot \big(2^i - 1\big) \cdot 2^{N-i} \qquad (6)$$

Equation (6) calculates the number of minterms in a Shadow Logic Function by subtracting the un-tainted minterms from all possible minterms in the shadow logic truth table. With this analysis, the number of minterms can be calculated for a varying number of input AND gates outlined in the experiments section.

Number of Minterms for OR_N: The Shadow Logic Function for OR_N is the dual of the logic function for AND_N and can be seen from the equation below.

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

As already shown, the function $f$and $\bar{f}$ 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. That is to say, the Shadow Logic Functions for an AND gate and OR gate have the same number of minterms due to this property. However, it is important to keep in mind that they do not have the same logic function. This can be taken a step further; any two functions f and g satisfying the following equation:

$$f(A_1, A_2, \cdots, A_n) = NOT\ g(\overline{A_1}, \overline{A_2}, \cdots, \overline{A_n})$$

have the same number of minterms in the Shadow Logic Function.

Number of Minterms for $NAND_N$ and NOR_N:

As stated in Theorem 2, invertors change logic values while keeping the taint status un-touched. So any two functions satisfying the following equation have exactly the same Shadow Logic Function.

$$f(A_1, A_2, \cdots, A_n) = NOT\ g(A_1, A_2, \cdots, A_n)$$

With this it can be seen that $NAND_N$ and NOR_N have the same number of minterms as AND_N and OR_N.

Number of Minterms for XOR_N:

To count the number of minterms in the Shadow Logic Function for XOR_N, the number of un-tainted entries from the shadowed truth table needs to be removed from all possible values. Since all input values need to be un-tainted 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 (7) formalizes this analysis:

$$num = 2^{2N} - 2^N \qquad (7)$$

The next section provides experimental results using this counting method for gate primitives. It also provides the number of minterms for more complicated combinational logic circuits including ISCAS benchmarks.

3. EXPERIMENTAL RESULTS

This section shows experimental results in terms of the number of minterms of a Shadow Logic Function. The number of minterms is used as our 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.

Quantitative calculations are done on a varying number of input AND, OR, NAND, NOR, and XOR gates using the counting method in Section III. Tests were also run on combinational ISCAS benchmarks and non-ISCAS benchmarks using the methods discussed in Section III.

3.1 Number of Minterms for Gate Primitives

Using the counting methods as addressed in Section III, the number of minterms were calculated for input sizes of 1 to 12 for AND, OR, NAND, NOR, and XOR. The results for these gates can be seen in Figure 3 below. The number of minterms corresponds to the amount of taint that a function will pass from the inputs to the outputs.

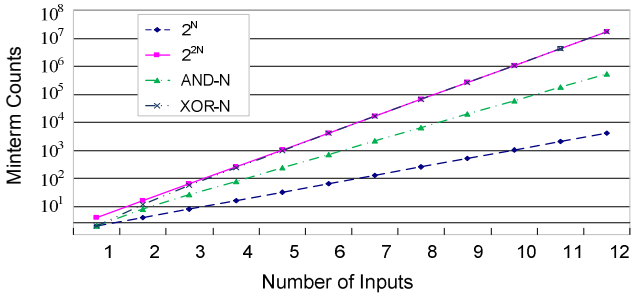

Figure 1—The Shadow Logic Function of gate primitives grows exponentially on the number of inputs. AND, OR, NAND, and NOR all have the same number of minterms. This plot is in log scale. The number of minterms corresponds to the amount of taint that propagates through the function.

As discussed in Section III, AND, OR, NOR, and NAND all share the same number of minterms due to Theorem 2. The results show that as the number of inputs increase, the number of minterms of the Shadow Logic Function 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 not be tainted in order for the result to be un-tainted.

3.2 Number of Minterms for ISCAS Benchmarks

The number of minterms for Shadow Logic Functions of some more complex logic circuits can also be realized. The results in Figure 4 are shown for ISCAS benchmarks 74L85(4-bit magnitude comparator), 74283(Adder), and s344/s349 (4x4 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 is seen in these more complicated logic blocks at it was seen for the gate primitives. Graphically these numbers are represented in Figure 4 below. The Figure is grouped into sections based on the logic blocks tested. Specifically 1, 2, 3, and 4 input Adders, Multipliers, and Comparators were tested as well as 2, 4, and 8 bit Shifters and Multiplexers. Figure 4 shows that, similar to the gate primitives, the number of minterms for more complex logic blocks also increases exponentially on the number of inputs, i.e. $O(2^{2N})$.

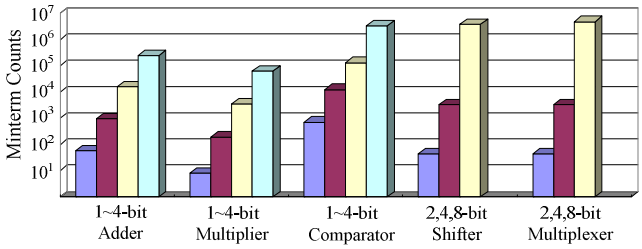

Figure 2—The Shadow Logic for more complicated logical functions also grows exponentially on the number of inputs.

This data shows that tracking data at the gate-level becomes difficult to manage because of the large increase in the number of minterms.

4. CONCLUSION

This paper presents the theory behind the Shadow Logic for systems that implement Gate Level Information Flow Tracking (GLIFT). It provides an analysis by formally defining taint and its properties. It also presents different methods for understanding shadow logic. Experimental results show that the number of minterms for the shadow logic of a logic function grows exponentially with the number of inputs $O(2^{2N})$.

5. REFERENCES

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

[2] A. Sabelfeld and A. C. Myers. "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[3] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[4] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. "Secure Program Execution via Dynamic Information Flow Tracking," In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[5] M. Dalton, H. Kannan, and C. Kozyrakis. "Raksha: A Flexible Information Flow Architecture for Software Security," In *34th Intl. Symposium on Computer Architecture (ISCA)*, June 2007.

[6] 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)*, 2009.

[7] 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," *Proceedings of the International Symposium on Microarchitecture (Micro)*, December 2009. New York, NY