Gate-Level Information Flow Tracking for Security Lattices

WEI HU and DEJUN MU, Northwestern Polytechnical University
JASON OBERG, University of California, San Diego
BAOLEI MAO, Northwestern Polytechnical University
MOHIT TIWARI, University of Texas, Austin
TIMOTHY SHERWOOD, University of California, Santa Barbara
RYAN KASTNER, University of California, San Diego

High-assurance systems found in safety-critical infrastructures are facing steadily increasing cyber threats. These critical systems require rigorous guarantees in information flow security to prevent confidential information from leaking to an unclassified domain and the root of trust from being violated by an untrusted party. To enforce bit-tight information flow control, gate-level information flow tracking (GLIFT) has recently been proposed to precisely measure and manage all digital information flows in the underlying hardware, including implicit flows through hardware-specific timing channels. However, existing work in this realm either restricts to two-level security labels or essentially targets two-input primitive gates and several simple multilevel security lattices. This article provides a general way to expand the GLIFT method for multilevel security. Specifically, it formalizes tracking logic for an arbitrary Boolean gate under finite security lattices, presents a precise tracking logic generation method for eliminating false positives in GLIFT logic created in a constructive manner, and illustrates application scenarios of GLIFT for enforcing multilevel information flow security. Experimental results show various trade-offs in precision and performance of GLIFT logic created using different methods. It also reveals the area and performance overheads that should be expected when expanding GLIFT for multilevel security.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Information Flow Controls

General Terms: Security, Design, Verification

Additional Key Words and Phrases: High-assurance system, hardware security, gate-level information flow tracking, multilevel security, security lattice, formal method

ACM Reference Format:
Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2014. Gate-level information flow tracking for security lattices. ACM Trans. Des. Autom. Electron. Syst. 20, 1, Article 2 (November 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2676548

1. INTRODUCTION

High-assurance systems such as those found in industrial infrastructures, financial systems, and medical devices are under ever-increasing risk of cyber attacks [Vishik

This work was supported by the NSF under grant CNS-11621776 and NSFC under grant 61303224.
Authors' addresses: W. Hu (corresponding author) and D. Mu, School of Automation, Northwestern Polytechnical University, 127 Youyi West Road, Beilin, Xian, Shaanxi, China; email: vinnie103@gmail.com; J. Oberg, Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093; B. Mao, School of Automation, Northwestern Polytechnical University, 127 Youyi West Road, Beilin, Xian, Shaanxi, China; M. Tiwari, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712; T. Sherwood, Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106; R. Kastner, Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093.

DOI: http://dx.doi.org/10.1145/2676548

et al. 2012]. For example, the smart grid is facing rapidly evolving cyber threats. The networking of grid control elements, connection of smart meters in customer premises, and integration of corporate billing systems mean that the smart grid can potentially be attacked remotely, without physical presence [Mo et al. 2012]. Modern intelligent cars are also exposing new security vulnerabilities, which allows attackers to remotely control safety-critical components such as the braking system [Checkoway et al. 2011]. In addition, there is published work demonstrating successful attacks on cardiac pacemakers [Halperin et al. 2008] and insulin pumps [Li et al. 2011] through wireless communication channels, which puts the patients' safety and privacy at risk. To address these newly emerging cyber threats, effective security measures and extensive design efforts should be collaborated from the early system design phase or otherwise high-assurance systems may be confronted with catastrophic consequences.

Two commonly used security techniques are *data encryption* and *access control*. Cryptographic algorithms are effective in enforcing data secrecy during data storage and transfer. However, they cannot protect sensitive information against leakage once it is decrypted for processing. Additionally, modern cryptographic algorithms heavily rely on secure key management. Even the strongest cryptographic algorithm will automatically fail if its execution environment leaks the private key through a covert channel. To prevent illegal access to confidential information, access control policies are usually deployed to restrict the access rights of unauthorized users. However, access control mechanisms (discretionary, mandatory, or role-based) have an inherent limitation. Specifically, they do prevent information from being accessed illegally, but cannot prevent it from being propagated improperly by authorized users. In addition, even in systems where access control is strictly enforced, it still could be possible to transmit information indirectly using system side-effects. Furthermore, new attack techniques tend to exploit security vulnerabilities, which can be far more efficient than cracking a cryptographic algorithm or breaking through an access control policy.

A complementary approach is to enforce tight *information flow control* (IFC). This approach classifies data objects into different security levels and monitors the propagation of data among security domains to prevent sensitive information from leakage or high-integrity data from being violated. IFC can be implemented either through static information flow analysis or dynamic information flow tracking (IFT) to enforce certain information flow security policies such as Bell LaPadula for confidentiality [Bell and LaPadula 1973] and noninterference [Goguen and Meseguer 1982] for integrity. A survey by Sabelfeld and Myers [2003] has summarized the extensive research in static information flow analysis at the programming language and compiler levels. Krohn et al. [2007] and Vandebogart et al. [2007] have built IFC mechanisms into standard OS primitives such as processes, pipes, and the filesystem for lower design complexity. Suh et al. [2004], Newsome and Song [2005], and Dalton et al. [2007] have implemented strict IFC at the ISA/uARCH level to reduce performance overheads. Although IFC has been shown effective in preventing harmful flows of information and detecting security vulnerabilities, the methods mentioned earlier are all at too high a level of abstraction to identify hardware-specific timing channels that have been shown to cause secret key leakage in stateful components such as caches [Bernstein 2005] and branch predictors [Acıiçmez et al. 2006].

To account for hardware-specific timing flows, Tiwari et al. [2009b] has recently proposed an IFT method called gate-level information flow tracking (GLIFT) that tracks the flow of information through Boolean gates. At such a low level of abstraction, all digital information flows, such as explicit flows, implicit flows, and even hardware-specific timing flows, appear in a mathematically unified form. This will allow detection of hardware-specific timing channels that are inherently invisible at higher levels

of abstraction. In previous work [Tiwari et al. 2009a], an execution lease architecture was developed to prevent undesired flows of information, including those through hardware-specific timing channels, to restrict the effects of untrusted programs to a constrained spatial and temporal boundary. This architecture employs GLIFT to show provable information flow isolation between different execution contexts. Oberg has shown how GLIFT can be used to identify and eliminate timing channels in shared buses architectures such as I2C, USB, and Wishbone [Oberg et al. 2011, 2013a]. Further, a practical testing framework is constructed to prove strict isolation between IP blocks with different trust basis in SoC (System-on-Chip) systems [Oberg et al. 2014]. Again, GLIFT is used to identify and eliminate harmful flows of information, including those through hard-to-detect timing channels.

The preceding preliminary work has demonstrated the effectiveness of GLIFT for enhancing information flow security, especially in detecting and eliminating hardware-specific timing channels. However, the GLIFT method employed in the aforesaid work targets two-level security labels, which is insufficient for enforcing multilevel security (MLS) [Denning 1976]. For example, military systems usually require an at least four-level security classification, namely `Unclassified`, `Confidential`, `Secret`, and `Top secret`, that cannot be modeled using a two-level linear security lattice. Another typical example can be found in SoC systems, where designers often need to prevent undesirable interference (caused by harmful flows of information) between IP cores of different trust (e.g., `Open-source`, `IP-vendor`, and `Self-developed`). A two-level linear security lattice simply cannot be used to model such information policies. In addition, many systems tend to be interested in nonlinear security lattices for proving isolation between incomparable entities. Thus, we need to expand GLIFT to more general security lattices in order to adapt to a wider range of systems.

To meet the requirements for MLS, we have made a first attempt to expand the GLIFT method to target multilevel security labels in Hu et al. [2013]. However, the paper focused on two-input primitive gates and several simple security lattices. As a result, it did not fully demonstrate how multilevel security labels are propagated through Boolean circuits. Specifically, label propagation automatically reduced to the case for a two-level security label since the primitive gates considered could take two inputs at most each time. In addition, a naive label propagation rule-set enumeration method was used to derive GLIFT logic for primitive gates. The complexity of such a enumeration method would soon become intractable when the lattice structure grows more complex. To reduce complexity, a constructive approach based on a minimum GLIFT library was proposed to augment tracking logic for primitive gates in large digital circuits discretely. However, such a constructive method has a potential imprecision problem, which will be addressed in successive discussions. Thus, the problem of GLIFT for MLS still remains unresolved.

This article provides a general way to expand the GLIFT method to meet the requirements for MLS. It presents the basic concepts, GLIFT logic formalizations, and generation method for the expansion. In addition, it also shows what sort of area and performance overheads should be expected when expanding GLIFT to target multiple levels of security labels. Specifically, this article makes the following contributions.

—*Providing a general way to expand GLIFT for MLS.* We provide an approach to expand the GLIFT method to target arbitrary levels of security labels for proving multilevel security.
—*Presenting generalized formalization of GLIFT logic for Boolean gates.* We derive symbolic representation of GLIFT logic for basic logical constructs (NOT, BUF, Flip-Flop, AND, NAND, OR, NOR, XOR, XNOR, and tri-state gates) under multiple levels of security labels.

—*Performing precision and complexity analysis of GLIFT logic.* We perform quantitative analysis of GLIFT logic for *Trust-Hub* [Baumgarten et al. 2011] and *IWLS* [2005] benchmarks in terms of precision, area, and performance.

The remainder of this article is organized as follows: Section 2 introduces the threat model as well as preliminaries of information flow analysis, with an emphasis on the basis of information flow security and related work in enforcing information flow control at various abstraction levels. In Section 3, we define some concepts and operations related to security lattices. We provide a general way to expand the GLIFT method to multilevel security labels in Sections 4 and 5, formalizing GLIFT logic for Boolean gates and presenting a precise GLIFT logic generation method. Section 6 shows how GLIFT can be used to enforce MLS, either through static information testing/verification or dynamic information flow tracking, and also addresses various design optimization considerations. In Section 7, we perform precision and complexity analysis of GLIFT logic circuits under several security lattices using Trust-Hub [Baumgarten et al. 2011] and IWLS [2005] benchmarks. Finally, we conclude this article in Section 8.

2. PRELIMINARIES

This section covers the threat model used in this article and some basic concepts of information flow analysis. First, we introduce our threat model and the different types of logical information flows in digital systems. Then, we briefly review the related work in IFT, a frequently used technique for enforcing information flow security. Finally, we present the latest research in GLIFT with a discussion on its limitations.

2.1. Threat Model

In this article, we account for security threats caused by vulnerabilities in hardware design that may result in violations of security properties of *integrity* or *confidentiality*. For integrity, design flaws that allow untrusted inputs to flow to high-integrity regions of the design are considered as a security threat (e.g., unprotected data from the open network interface being used to manipulate the program counter). For confidentiality, security holes that cause sensitive information to leak to an unclassified domain are considered as a security threat (e.g., a private key in a cryptographic core flowing to outputs other than the cipher text). Such security property violations are associated with harmful flows of information between different security domains. Thus, these security threats can be modeled as information flow security policy violations.

In our analysis, we target the abstraction level of Boolean gates and focus on logical information flows that propagate through digital hardware. Our method can be used to detect potential security vulnerabilities in hardware design by capturing harmful flows of information, or to enforce strict information flow isolation between components of different security classification, such as third-party cores and high-integrity cores built in house. It does not aim to handle attacks that require statistical analysis in order to retrieve secret information or account for information flows caused by physical phenomena, such as power dynamics or electromagnetic radiations, that do not appear at the logical level.

2.2. Logical Information Flows

Logical information flows in digital systems can be categorized into *explicit* and *implicit flows*.

The simplest case is the explicit flow, which is always associated with direct data movements. Thus, explicit flow is also called *dataflow*. Examples of explicit flows can be found in evaluation expressions where information flows from source to destination

operand and in network communications where information flows from sender to receiver. In the following example, information flows explicitly from `secret` to `leak`.

```
TYPE_SECRET secret;
TYPE_UNCLASSIFIED leak;
leak := secret;
```

A more subtle case is the implicit flow, caused by nondeterministic system behaviors such as conditional branching or latency. Correspondingly, implicit flow can alternatively be called *control flow*. For example, the latency difference between cache hit and miss creates an implicit flow that could cause secret key leakage [Bernstein 2005]. In the following example, the least significant bit of `secret` flows implicitly to `leak`.

```
TYPE_SECRET secret;
TYPE_UNCLASSIFIED leak;
IF (secret & 0x01) THEN
  leak := TRUE;
ELSE
  leak := FALSE;
```

Timing flow is a special type of implicit flow that transmits information through timing-related behaviors. In the following example, the attacker can deduce whether `secret` is nonzero by observing the execution time of the program.

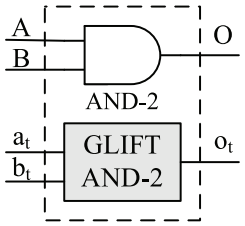
```
TYPE_SECRET secret;
TYPE_UNCLASSIFIED done = FALSE;
IF (secret == TRUE) THEN
  heavy_computation();
done := TRUE;
```

From the prior examples, we can see that undesirable flows may leak secret information. Thus, digital systems should be designed with careful consideration of both explicit and implicit flows and effective measures taken to prevent sensitive information from leakage, or, the dual, high-integrity data from being violated. In practice, this can be achieved through tight IFC. In the following section, we will briefly discuss the related work in IFT that is commonly used for implementing IFC.

2.3. Information Flow Tracking

In IFT, data are assigned a label to indicate their security levels (e.g., `trusted` or `untrusted`). The label is propagated along with data through the system. IFT monitors the propagation of information to check whether secret data leak to an unclassified domain or high-integrity data are violated by an untrusted party.

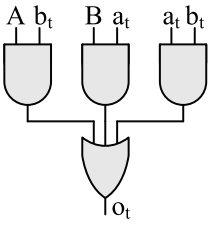
Most IFT methods focus on tracking information flows at the program language (PL), operating system (OS), instruction set architecture (ISA), and microarchitecture (μARCH) levels. PL-level IFT methods enforce information flow security through compile-time static verification with the employment of typing systems [Volpano et al. 1996; Pottier and Simonet 2003]. Although these methods introduce little overhead in the final implementations, they force programmers to comply with new typing systems that lead to higher design complexity. OS-level IFT methods monitor information flows with abstractions for operating system primitives such as processes, pipes, and the filesystem [Krohn et al. 2007; Vandebogart et al. 2007]. They build IFC mechanisms into the OS and thus can take the pressure of high design complexity off the programmers. However, these methods typically report up to 30% performance overhead and, like PL-level methods, are at too high a level of abstraction to capture hardware-specific timing channels. ISA/μARCH-level IFT implementations [Suh et al. 2004; Newsome



(a)

A	a_t	B	b_t	O	o_t
0	0	0	1	0	0
0	0	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	1

(b)



(c)

Fig. 1. The GLIFT method. (a) I/O of AND-2 and its GLIFT logic; (b) partial truth table of GLIFT logic for AND-2; (c) GLIFT logic of AND-2.

and Song 2005; Dalton et al. 2007] track information flows at the granularity of instruction and data words. They introduce very low performance overhead but are also at a level of abstraction that is transparent to timing behaviors in the underlying hardware.

In addition, these IFT methods tend to be conservative in calculating the security labels for the output, since they only consider the security label of the inputs and ignore the *value* they actually take. It is often the case that a higher security-level input does not essentially have an effect on the output even if it is involved in a computation. Specifically, consider n data objects $A_1, A_2, \ldots, A_n$ with security labels $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n$, respectively. When an operation is performed on these objects, the security label of the output will be assigned to the least upper bound of $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n$ in previous IFT methods. This is secure but overly conservative because information contained in $A_1, A_2, \ldots, A_n$ may not necessarily all flow to the output according to information theory [Shannon 2001].

While information flows appear in various forms at the PL, OS, ISA, and μARCH levels, they all flow explicitly in the granularity of binary bits at the gate level and thus can be precisely defined in a form that unifies the notions of explicit flows, implicit flows, and even hardware-specific timing flows [Oberg et al. 2011]. Recently, gate-level information flow tracking (GLIFT) has been proposed to precisely measure and manage all digital flows from the level of Boolean gates [Tiwari et al. 2009b].

2.4. Gate-Level Information Flow Tracking

In GLIFT, each binary data bit is associated with a label to indicate its security level, such as trusted/untrusted or confidential/unclassified. GLIFT provides a more precise approach to IFT in that the output is bounded to the most restrictive security label whose data actually affects the output. For a better understanding, consider the AND-2 example as shown in Figure 1(a), where A, B, and O are the inputs and output of AND-2, and a_t, b_t, and o_t are the security labels of A, B, and O, respectively. Let an I/O be untrusted when its security label is logical "1".

Consider the partial truth table in Figure 1(b), where A is trusted while B is untrusted. In the first row, both A and B have an influence at (or dominate) the output; thus, the output should be set to trusted since trusted is a more restrictive label. In the second and third rows, A (or B) determines the output; the output should take the security label of A (or B). In the fourth row, neither A nor B dominates the output. In this case, the output should be labeled as untrusted, since a change in the untrusted input B could lead to a change in the output. When considering a full truth table, we can derive the GLIFT logic for AND-2 as shown in Figure 1(c). We can see that the GLIFT logic takes into account both the data value and its security label while calculating the security label for the output. By comparison, previous IFT methods tend to ignore the actual influence of the data value at the output and always mark the output as

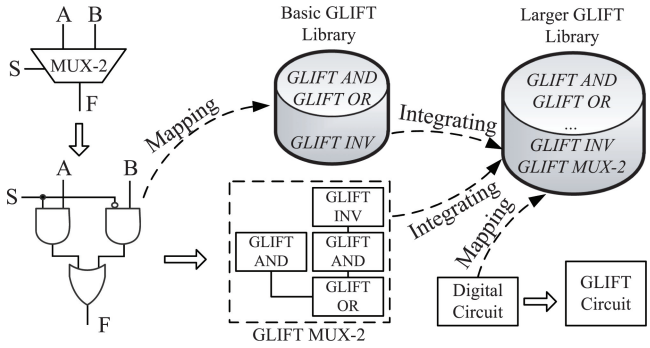


Fig. 2. The constructive method for GLIFT logic generation.

untrusted now that there is an untrusted input. Thus, GLIFT provides a more precise approach to IFT than previous conservative methods.

Previous work has demonstrated the effectiveness of GLIFT for building information-flow-secure architectures [Tiwari et al. 2009a,b, 2011; Kastner et al. 2011], especially in detecting hardware-specific timing channels [Oberg et al. 2011, 2013a, 2013b]. However, the aforesaid GLIFT method only targets two-level security labels. Real systems usually require or benefit from MLS policies that need to be modeled using multilevel security lattices [Denning 1976] (e.g., for modeling a security policy that specifies allowable information flows among IP cores from vendors of varying trust). In the following section, we will discuss the latest research in expanding GLIFT for MLS as well as its drawbacks.

2.5. GLIFT for Multilevel Security

To satisfy the requirements for MLS, we have made an initial attempt to expand GLIFT to target multiple levels of security labels in Hu et al. [2013]. However, expanding GLIFT to cope with multilevel security labels is a complicated process due to the inherently high complexity of fine-granularity IFT methods.

Without loss of generality, consider a digital circuit with n-bit inputs under a lattice with m security labels. For each input, the original variable can take two possible binary values while its security label has m alternatives. When considering all n inputs, their values have a total number of 2^n possible combinations and their security labels have m^n possible input patterns. Thus, the complexity of the GLIFT method can be formalized as in (1). As an example, consider the GLIFT method under two-level security labels, namely, $m = 2$. The complexity of the method reduces to $O(2^{2n})$, which agrees with the theoretical analysis in Hu et al. [2011].

$$O(2^n \cdot m^n) = O((2m)^n) \tag{1}$$

From (1), the general GLIFT problem inherently has exponential complexity. To solve the problem in polynomial time, we set n to be a constant and restrict our discussions to two-input primitive gates (i.e., $n = 2$, in Hu et al. [2013]). With such a restriction, the complexity of the problem reduces to $O((2m)^2)$. Subsequently, GLIFT logic for large digital circuits is created in a constructive manner. In this constructive method, a functionally complete GLIFT library containing the tracking logic for primitive gates is maintained. Given a digital circuit, it is synthesized to a gate-level netlist composed of the primitive gates found in the GLIFT library. Then, one can discretely instantiate tracking logic for each primitive gate in the netlist through a constant-time mapping operation. Figure 2 illustrates such a constructive approach.

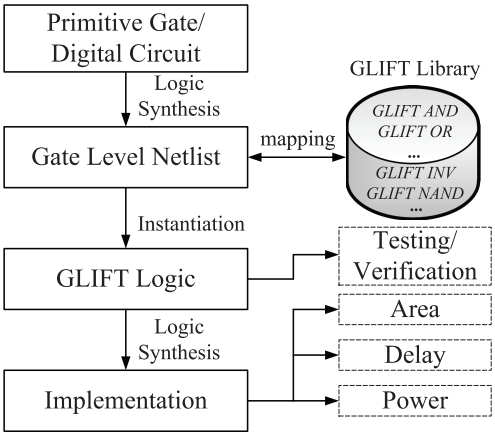


Fig. 3. The design flow of the constructive method.

As shown in Figure 2, a minimum GLIFT library consisting of the tracking logic for AND, OR, and INV (inverter) is maintained initially. This GLIFT library is functionally complete in describing all Boolean circuits. Given a two-to-one multiplexer (MUX-2), it is synthesized into a netlist with two AND gates, an OR gate, and an INV. Subsequently, the GLIFT logic for MUX-2 can be obtained by mapping the netlist to the minimum GLIFT library. Once the GLIFT logic for MUX-2 is created, it can be integrated with the basic GLIFT library to form a larger library. In this way, more complex GLIFT libraries can be constructed and tracking logic for large digital circuits can be generated constructively.

Figure 3 shows the design flow of the constructive method. There can be various optimizations like those in technology mapping during GLIFT logic instantiation. However, we currently rely on logic synthesis tools for design optimization, which are performed during a first synthesis of the original design to a gate-level netlist and a second synthesis of the resulting GLIFT logic for final implementation. In our future work, we will investigate various optimizations during the mapping of primitive gates to the GLIFT library in order to reveal the effect of different ways of mapping on area, performance, and precision.

The constructive method could be highly efficient for GLIFT logic generation since it has polynomial time complexity without considering the logic synthesis process. However, we have observed that GLIFT logic generated using this method may contain *false positives*, indicating nonexistent flows of information; this will be addressed in detail in Section 5.1.

This article intends to provide a general way in which GLIFT can be expanded to target multilevel security labels. It presents a solution to this complex expansion problem by formalizing tracking logic for primitive Boolean gates under multilevel security labels and presenting a method for precise[1] GLIFT logic generation. Before this, we define some terms and definitions.

3. TERMS AND DEFINITIONS

Denning [1976] first proposed to use the lattice model for describing information flow policies. To facilitate successive discussions, we restate some essential concepts for the lattice model [Denning 1982] and define some operations on security labels.

[1]Precise GLIFT logic indicates logic TRUE when and only when there is an actual information flow.

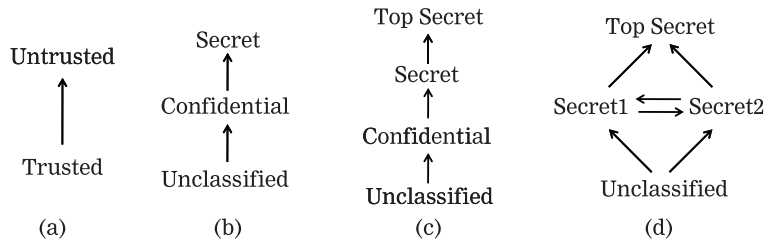


Fig. 4. Sample security lattices. (a) Two-level linear lattice for integrity; (b) three-level linear lattice for confidentiality; (c) four-level linear lattice for confidentiality; (d) a square lattice for confidentiality.

Definition 3.1 (*Lattice*). Given a partial order set $\mathcal{L} = \{\mathcal{S}, \sqsubseteq\}$, where $\mathcal{S}$ is the set of elements and $\sqsubseteq$ is a partial order relation defined on the element set, the two-tuple $\{\mathcal{S}, \sqsubseteq\}$ constitutes a *lattice* if there is a least upper bound (*lub*) element and a greatest lower bound (*glb*) element for any $a, b \in \mathcal{S}$. Let $\oplus$ and $\odot$ be the least upper and greatest lower bound operators, respectively, these are denoted as

$$lub(a, b) = a \oplus b$$
$$glb(a, b) = a \odot b.$$

Definition 3.2 (*Maximum and Minimum Elements*). Let $\mathcal{S} = \{a_1, a_2, \ldots, a_n\}$ be the element set of a given lattice. We define the *maximum element* (denoted as `HIGH`) and *minimum element* (denoted as `LOW`) on the lattice as follows[2].

$$HIGH = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$
$$LOW = a_1 \odot a_2 \odot \cdots \odot a_n$$

Definition 3.3 (*Security Lattice*). A *security lattice* is a lattice whose element set is composed of security classes. A given information flow policy can be modeled using a security lattice $\mathcal{L} = \{\mathcal{SC}, \sqsubseteq\}$, where $\mathcal{SC}$ is a set of security classes and $\sqsubseteq$ is the partial order defined on $\mathcal{SC}$.

Figure 4 shows some simple security lattices. The set of security classes is a combination of all possible security levels that data objects can take (e.g., $\mathcal{SC} =$ {`Unclassified`, `Confidential`, `Secret`} for a three-level linear confidentiality lattice as shown in Figure 4(b)). The arrows show the permissible directions of information flows and reflect the partial order defined by the lattice. Let $\mathcal{L} : \mathcal{O} \rightarrow \mathcal{SC}$ be a function that returns the security class of an object in $\mathcal{O}$. When $\mathcal{L}(A) \sqsubseteq \mathcal{L}(B)$, information flowing from A to B will not violate the security policy specified by the lattice and thus is secure. Generally, an information flow security policy only allows information to flow within the same security class or upward along the security lattice. Any downward information flow will cause a security policy violation. Given two security classes a and b, we define a to be more restrictive than b (or b to be more conservative than a) when $a \sqsubseteq b$. With an understanding of the partial order on security lattices, we can define and prove the following operation laws for comparable security classes.

Proposition 3.4 (Absorption Law).

$$A_t \odot B_t \oplus A_t \odot B_t \odot C_t = A_t \odot B_t \tag{2}$$

Proof. According to the definition of least upper bound,

$$A_t \odot B_t \sqsubseteq A_t \odot B_t \oplus A_t \odot B_t \odot C_t. \tag{3}$$

[2]`HIGH` and `LOW` correspond to `Confidential` and `Unclassified` in confidentiality analysis, and to `Untrusted` and `Trusted` in integrity analysis.

Also, according to the definition of greatest lower bound,

$$A_t \odot B_t \odot C_t \sqsubseteq A_t \odot B_t, \tag{4}$$

therefore

$$A_t \odot B_t \ \oplus \ A_t \odot B_t \odot C_t \sqsubseteq A_t \odot B_t \ \oplus \ A_t \odot B_t = A_t \odot B_t. \tag{5}$$

With (3) and (5), the Absorption Law holds. □

PROPOSITION 3.5 (DISTRIBUTIVE LAW).

$$(A_t \ \oplus \ B_t) \odot C_t = A_t \odot C_t \ \oplus \ B_t \odot C_t \tag{6}$$

PROOF. Since A_t and B_t are comparable, assume $A_t \sqsubseteq B_t$ without loss of generality,

$$(A_t \oplus B_t) \odot C_t = B_t \odot C_t. \tag{7}$$

According to the definition of least upper bound,

$$B_t \odot C_t \sqsubseteq A_t \odot C_t \ \oplus \ B_t \odot C_t. \tag{8}$$

Since $A_t \sqsubseteq B_t$, we have

$$A_t \odot C_t \ \oplus \ B_t \odot C_t \sqsubseteq B_t \odot C_t \ \oplus \ B_t \odot C_t = B_t \odot C_t, \tag{9}$$

and with (8) and (9), the Distributive Law holds. □

Similarly, we can prove the following *Associative Law*.

PROPOSITION 3.6 (ASSOCIATIVE LAW).

$$A_t \odot C_t \ \oplus \ B_t \odot C_t = (A_t \ \oplus \ B_t) \odot C_t \tag{10}$$

In addition, the greatest lower bound operator needs to be redefined for incomparable security classes. Given two incomparable security labels A_t and B_t, their greatest lower bound should be no more restrictive than A_t or B_t. As an example, considering security classes `Secret1` and `Secret2` in Figure 4(d), `Secret1` $\odot$ `Secret2` should be set to `Secret1` or `Secret2` (or even conservatively to `Top Secret`) instead of `Unclassified`. From the example, the result of the greatest lower bound operation on incomparable security classes can be nondeterministic. Thus, we redefine the greatest lower bound operator on incomparable security classes as (11). Specifically, we construct a candidate set consisting of all the possible safe output security classes and choose the most restrictive one from the candidate set as the output label. We randomly pick one if there are still multiple choices (e.g., `Secret1` and `Secret2` shown in Figure 4(d)) that are symmetric.

$$A_t \odot B_t \in \{C_t \mid C_t \sqsubseteq \ S, \ \forall \ A_t \sqsubseteq \ S \ and \ B_t \sqsubseteq \ S\} \tag{11}$$

For simplicity, we will use a unified notation for the greatest lower bound operator in our discussion. Eq. (11) should be used when calculating greatest lower bounds for incomparable security classes.

Definition 3.7 (*Dot Product*). We define the *dot product* on a Boolean variable A and a security label vector B_t as (12). Such a dot product operation implies logical AND when applied to Boolean variables.

$$A \cdot B_t = \begin{cases} LOW & A = 0 \\ B_t & A = 1 \end{cases} \tag{12}$$

Using (12), we can verify the following Associative Law on the dot product operator by assigning A to "0" and "1", respectively, as

$$(A \cdot B_t) \odot C_t = A \cdot (B_t \odot C_t), \tag{13}$$

where A is a Boolean variable while B_t and C_t are security label vectors.

Eq. (14) defines the Distributive Law on the dot product operator, where the symbol $\vee$ denotes logical OR operation. The law can be proved by assigning A or B to "0" and "1", respectively.

$$(A \vee B) \cdot C_t = A \cdot C_t \ \oplus \ B \cdot C_t \tag{14}$$

We can derive the following Absorption Law by replacing B with AB in (14).

$$A \cdot C_t \ \oplus \ AB \cdot C_t = A \cdot C_t \tag{15}$$

Without loss of generality, we consider an arbitrary security lattice $\mathcal{L} = \{\mathcal{SC}, \sqsubseteq\}$ in our successive discussions. Let $m = |\mathcal{SC}|$ be the total number of security classes. Then the security label for a Boolean variable needs to be denoted with at least $w = \lceil log_2 m \rceil$ binary bits. We use upper-case letters with/without a superscript to denote Boolean variables (e.g., A, B, A^1, A^2, ..., A^n), while their security labels are denoted as A_t, B_t, A_t^1, A_t^2, ..., A_t^n correspondingly. Note that each security label is a w-dimensional vector, for instance, $A_t = (a_t^{w-1}, \ldots, a_t^1, w_t^0)$. The logical AND ($\wedge$) and dot product operators will be eliminated wherever possible for simplicity.

With the preceding notations and definitions, we will provide a general way to expand GLIFT for MLS. A fundamental task in this expansion process is GLIFT logic generation. Thus, we first formalize GLIFT logic for Boolean gates in Section 4 and then provide a method for false-positive-free GLIFT logic generation in Section 5.

4. FORMALIZING GLIFT LOGIC FOR BOOLEAN GATES UNDER MULTILEVEL SECURITY LATTICES

4.1. Buffers, Inverters and Flip-Flops

A common property of buffers, inverters, and flip-flops is that a change in the input will always be reflected at the output, indicating an information flow. Although the value observed at the output may be inverted, these components never lead to a change in the security label. Let I denote the input and O the output. The GLIFT logic for these components can be formalized as (16).

$$O_t = I_t \tag{16}$$

It should be noted that the GLIFT logic for flip-flops has some slight difference, since flip-flops are sequential components where signal transition activities usually take place at clock edges. Thus, they will delay the propagation of security labels for one clock cycle.

4.2. AND/NAND Gates

Consider the two-input AND gate (AND-2) whose Boolean function is $O = A \wedge B$. To formalize GLIFT logic under multilevel security lattices, we need to expand the security labels to multidimensional vectors and apply the corresponding operators on security label vectors. Specifically, `dot product` should be used to imply multiplication of a Boolean variable with a security label; the least upper and greatest lower bound operators need to be used for operations on security labels. Table I shows the truth table that calculates the output security label of AND-2, where A_t, B_t, and O_t are the security label vectors of A, B, and O, respectively.

As an example, consider the case when $A = 0$ and $B = 1$. In this case, O_t should be set to A_t since A dominates the output. Now consider the case when $A = 1$ and $B = 0$. In this case, O_t evaluates to B_t, indicating information flow from B to the output. More subtle cases could happen when A and B are both logical "1" or logical "0". Specifically, when A and B are both logical "1", neither A nor B has a direct influence on the output and the security class of the output will be evaluated to $A_t \oplus B_t$ (the more conservative one). When A and B are simultaneously logical "0", both A and B have an influence

Table I. Truth Table of GLIFT Logic for AND-2 under Multilevel Security Lattices

#	A	B	A_t	B_t	O	O_t
1	0	0	A_t	B_t	0	$A_t \odot B_t$
2	0	1	A_t	B_t	0	A_t
3	1	0	A_t	B_t	0	B_t
4	1	1	A_t	B_t	1	$A_t \oplus B_t$

on the output. The security class of the output will be evaluated to $A_t \odot B_t$ (the more restrictive one).

From Table I, we can obtain the GLIFT logic for AND-2 as shown in (17).

$$O_t = AB_t \ \oplus \ BA_t \ \oplus \ A_t \odot B_t \tag{17}$$

To formalize GLIFT logic for multiple input Boolean gates under multilevel security lattices, we start from considering the three-input AND gate (AND-3) under a three-level linear security lattice. Let the Boolean function of AND-3 be $O = A \wedge B \wedge C$. We can derive the GLIFT logic for AND-3 step by step using (17) as

$$O_t = C \cdot \mathcal{L}(AB) \ \oplus \ ABC_t \ \oplus \ \mathcal{L}(AB) \odot C_t, \tag{18}$$

where $\mathcal{L}(AB)$ denotes the security label of $A \wedge B$ as already formalized in (17).

Once (18) is expanded and simplified, we have

$$\begin{aligned} O_t = {} & ABC_t \ \oplus \ AC\,B_t \ \oplus \ BC\,A_t \ \oplus \ AB_t \odot C_t \\ & \oplus BA_t \odot C_t \ \oplus \ C\,A_t \odot B_t \ \oplus \ A_t \odot B_t \odot C_t. \end{aligned} \tag{19}$$

Now consider an n-input AND expression $O = A^1 \wedge A^2 \wedge \cdots \wedge A^n$ under an m-level security lattice. A possible solution is to evaluate the following polynomial and then apply appropriate operators, specifically the least upper and greatest lower bound operators on security label vectors. The minus sign in (20) denotes removing the term $A^1 A^2 \cdots A^n$ from the resulting equation.

$$O_t = \left(A^1 + A_t^1\right)\left(A^2 + A_t^2\right) \cdots \left(A^n + A_t^n\right) - A^1 A^2 \cdots A^n \tag{20}$$

With the tracking logic for AND gates, one can quickly derive GLIFT logic for NAND gates. Specifically, NAND and AND gates share the same tracking logic according to Section 4.1. To compose a minimum GLIFT library that is functionally complete in describing all digital circuits, the OR gate is usually included. In the following section, we derive GLIFT logic for OR gates under multilevel security lattices.

4.3. OR/NOR Gates

We first consider the two-input OR gate (OR-2) whose Boolean function is $O = A \vee B$. Using the DeMorgan Law [Maini 2007], we have

$$O = \overline{\overline{A} \wedge \overline{B}}.$$

According to Section 4.1, OR-2 has the same GLIFT logic as $\overline{A} \wedge \overline{B}$, which can be directly derived from (17) and is shown in (21).

$$O_t = \overline{A}B_t \ \oplus \ \overline{B}A_t \ \oplus \ A_t \odot B_t \tag{21}$$

More generally, consider an n-input OR expression $O = A^1 \vee A^2 \vee \cdots \vee A^n$. A feasible solution is to evaluate the following polynomial and then apply appropriate operators on security label vectors, namely, $\oplus$ for plus while $\odot$ for product. The minus sign in (22) denotes removing the term $\overline{A^1}\ \overline{A^2} \cdots \overline{A^n}$ from the resulting equation.

$$O_t = \left(\overline{A^1} + A_t^1\right)\left(\overline{A^2} + A_t^2\right) \cdots \left(\overline{A^n} + A_t^n\right) - \overline{A^1}\ \overline{A^2} \cdots \overline{A^n} \tag{22}$$

Similarly, NOR and OR gates share the same GLIFT logic as well, according to Section 4.1. By (20) and (22), the GLIFT logic for OR gates can be quickly derived from that for AND gates by merely inverting the Boolean variables and vice versa.

4.4. XOR/XNOR Gates

The XOR and XNOR are a special type of gate in that they are information flow sensitive. Specifically, whenever there is a change in the input, the change will be observed at the output. With such an understanding, the GLIFT logic for n-input XOR and XNOR gates with inputs $A^1, A^2, \ldots, A^n$ can be described as (23).

$$O_t = A_t^1 \oplus A_t^2 \oplus \cdots \oplus A_t^n \tag{23}$$

4.5. The Tri-State Gate

The tri-state gate is another special type of logic construct commonly found in digital circuits. The output of the tri-state gate is selected between the input and high-impedance state by a control signal. The logic function of the tri-state gate can be described as (24), where S, I, O, and "Z" represent the control signal, input, output, and high-impedance state, respectively.

$$O = S\ ?\ I\ :\ 'Z'; \tag{24}$$

From (24), when S is asserted, the output O will be determined by I. Thus, the term SI_t should be added to the GLIFT logic to track information flow from I to O when $S = 1$. Similarly, when S is negated, the output O will be in high-impedance state. Therefore, the term $\overline{S} \cdot LOW$ should be added to the GLIFT logic since the security class for constants is `LOW`. Additionally, the control signal S is also information flow sensitive. Specifically, a change in S will always cause the output to switch between the logical "0"/"1" and high-impedance states. To model such transition activities, the term S_t should be included in the GLIFT logic. According to our analysis, the GLIFT logic for the tri-state gate can be formalized as (25).

$$\begin{aligned} O_t &= SI_t \oplus \overline{S} \cdot LOW \oplus S_t \\ &= SI_t \oplus S_t \end{aligned} \tag{25}$$

When the tri-state gate is used for driving shared buses, there can be some slight difference its GLIFT logic. Specifically, the tracking logic should be set to high impedance when the select line S is negated to prevent multiple driving sources. Eq. (26) shows the tracking logic for the tri-state gate in shared bus architectures.

$$\begin{aligned} O_t &= S\ ?\ (SI_t \oplus S_t)\ :\ 'Z' \\ &= S\ ?\ (I_t \oplus S_t)\ :\ 'Z' \end{aligned} \tag{26}$$

In addition, Boolean operators can also be used to set or reset registers, that is, assigning constant values to registers. As an example, $R = R\ xor\ R$ clears the register R. In such cases, the security label of the registers should be set to `LOW` since constants always have a `LOW` label.

Now that we have derived GLIFT logic for the basic building blocks of Boolean circuits, we will address the potential imprecision problem with the constructive method and then provide a method for precise GLIFT logic generation under multilevel security lattices in the following sections.

5. PRECISE GLIFT LOGIC GENERATION UNDER MULTILEVEL SECURITY LATTICES

5.1. Impreciseness of the Constructive Method

The constructive method can be used to generate GLIFT logic for digital circuits in linear time through a constant-time mapping operation. However, this constructive

approach has a potential imprecision problem. Specifically, GLIFT logic generated using this method may contain false positives, indicating nonexistent flows of information. For a better understanding, consider the MUX-2 whose Boolean function is $F = SA \vee \overline{S}B$ [Maini 2007]. Using (16), (17), and (21), we have

$$F_t = \overline{SA} \cdot \mathcal{L}(\overline{S}B) \oplus \overline{\overline{S}B} \cdot \mathcal{L}(SA) \oplus \mathcal{L}(SA) \odot \mathcal{L}(\overline{S}B). \tag{27}$$

When (27) is expanded and simplified using (2), (10), and (13) to (15), the GLIFT logic for MUX-2 can be formalized as (28).

$$F_t = SA_t \oplus \overline{S}B_t \oplus \overline{A}BS_t \oplus A\overline{B}S_t \oplus A_t \odot S_t \oplus B_t \odot S_t \oplus ABS_t \tag{28}$$

Considering the case when $A = 1, A_t = \texttt{LOW}$, $B = 1, B_t = \texttt{LOW}$, we have $F_t = S_t$, indicating that information flows from S to F. Actually, the output will be constantly class LOW in this case since $A_t = B_t = \texttt{LOW}$. Thus, the term ABS_t is a false positive that indicates nonexistent flow of information. In the following section, we present a feasible solution to eliminating such false positives.

5.2. Precise GLIFT Logic Generation Method

According to switching circuit theory, false positives in the GLIFT logic created by the constructive method are caused by static logic hazards resulting from single-variable switches. As proved by Eicherberger [1965], a Boolean function with all its prime implicants will be free of all static logic hazards. This provides a possible solution to eliminating false positives in GLIFT logic generated in a constructive manner. For a better understanding, consider the MUX-2. To include all prime implicants, a redundant term AB needs to be appended to its Boolean function. We denote the new function as $G = F \vee AB$, where $F = SA \vee \overline{S}B$. Eq. (29) shows the GLIFT logic for G generated using the constructive method.

$$G_t = \overline{F} \cdot \mathcal{L}(AB) \oplus \overline{AB} \cdot F_t \oplus \mathcal{L}(AB) \odot F_t \tag{29}$$

When $A = 0$, G_t can be simplified to (30).

$$G_t|_{A=0} = SA_t \oplus \overline{S}B_t \oplus BS_t \oplus A_t \odot S_t \oplus B_t \odot S_t \tag{30}$$

Similarly, when $A = 1$, G_t can be simplified to (31).

$$G_t|_{A=1} = SA_t \oplus \overline{S}B_t \oplus \overline{B}S_t \oplus A_t \odot S_t \oplus B_t \odot S_t \tag{31}$$

Thus, we have

$$\begin{aligned} G_t &= \overline{A} \cdot G_t|_{A=0} \oplus A \cdot G_t|_{A=1} \\ &= SA_t \oplus \overline{S}B_t \oplus \overline{A}BS_t \oplus A\overline{B}S_t \oplus A_t \odot S_t \oplus B_t \odot S_t. \end{aligned} \tag{32}$$

With a comparison to (28), we can discover that the additional term ABS_t is automatically reduced. We can further verify that all the terms in G_t now precisely measure actual information flows to the output. In other words, false positives are eliminated by adding the prime implicant AB.

For a more concrete understanding, observe (29), where $\mathcal{L}(AB)$ does not contain any false positives since there is no single-variable switch (e.g., S and $\overline{S}$ in F composes a single-variable switch). When $A = 1, A_t = \texttt{LOW}$, $B = 1, B_t = \texttt{LOW}$, we have $F = 1$ and $\mathcal{L}(AB) = \texttt{LOW}$. In this case, AB and $\mathcal{L}(AB)$ together dominate the output of (29), and G_t will be evaluated to LOW regardless of the output status of F_t. Further, GLIFT logic can only contain false positives and never indicates any false negatives [Hu et al. 2011]. G_t should be able to precisely track all the actual information flows in F, since G is

logically equivalent to F. Thus, adding the prime implicant AB eliminates the false positives while retaining all the actual information flows in F_t.

From the MUX-2 example, precise GLIFT logic can be generated using the constructive method now that the Boolean function contains all its prime implicants. However, finding all prime implicants has been proven an NP-hard problem [Palopoli et al. 1999]. To eliminate all logic hazards while retaining acceptable computational complexity, Lin and Devadas [1995] proposed a method that allows synthesis of static-logic-hazard-free circuits using BDDs (Binary Decision Diagrams). This provides a new method for precise GLIFT logic generation as shown in Algorithm 1.

ALGORITHM 1: The BDD Algorithm

```
   Input: Boolean function f(x1, x2, ..., xn) with inputs x1, x2, ..., xn
   Input: GLIFT library consisting of the precise GLIFT logic for MUX-2
   Output: Precise GLIFT logic of f denoted as sh(f)
1  f_BDD ← f, constructing a reduced ordered/free BDD from f
2  f_MUX-2 ← f_BDD, represent each node in the BDD using a MUX-2
3  for each MUX-2 node m ∈ f_MUX-2 do
4      if m has constant input(s) then
5          simplify MUX-2 node m to Boolean equation
6      else
7          map m to the GLIFT library // instantiate GLIFT logic for m
8          add GLIFT logic of m to sh(f)
9      end
10 end
11 return sh(f)
```

In this BDD method, a reduced ordered or free BDD [Lin and Devadas 1995] is first constructed from a given Boolean function (line 1). Then the BDD is converted into a MUX-2 network (line 2). If a MUX-2 node in the network has constant input(s) (line 4), it can be further reduced to a Boolean equation (line 5). Finally, the simplified MUX-2 network is augmented with GLIFT logic using the constructive method introduced in Section 5.1 (lines 7 and 8). Now that the GLIFT logic for MUX-2 is false positive free, the resulting GLIFT circuit will be precise.

For a better understanding, consider the Boolean function $F = AB \vee \overline{B}C \vee \overline{A}\,\overline{C}$. As shown in Figure 5(a), a reduced ordered BDD is first constructed. Then, the BDD is converted into a MUX-2 network as shown in Figure 5(b). Further, the two multiplexer nodes with constant inputs are reduced to $\overline{C}$ and C, respectively. Figure 5(c) shows the MUX-2 network after simplification. Finally, GLIFT logic is generated constructively by augmenting tracking logic for each MUX-2 node. In this step, the false-positive-free GLIFT logic for MUX-2 given in (32) is instantiated and the security labels for constant inputs should be set to `LOW`.

Eqs. (33) and (34) give the GLIFT logic for F created using the constructive and BDD methods, respectively, where *xnor* and *xor* are the logical Exclusive-NOR (*xnor*) and Exclusive-OR (*xor*) operators.

$$\begin{aligned} O_t = {} & (B\ xnor\ C)A_t \oplus (A\ xor\ C)B_t \oplus (A\ xor\ B)C_t \oplus A_t \odot B_t \\ & \oplus A_t \odot C_t \oplus B_t \odot C_t \oplus \boxed{\overline{A}\,\overline{B}C_t \ \oplus\ AC\,B_t \ \oplus\ B\overline{C}\,A_t} \end{aligned} \tag{33}$$

$$\begin{aligned} O_t = {} & (B\ xnor\ C)A_t \oplus (A\ xor\ C)B_t \oplus (A\ xor\ B)C_t \oplus A_t \odot B_t \\ & \oplus A_t \odot C_t \oplus B_t \odot C_t \end{aligned} \tag{34}$$

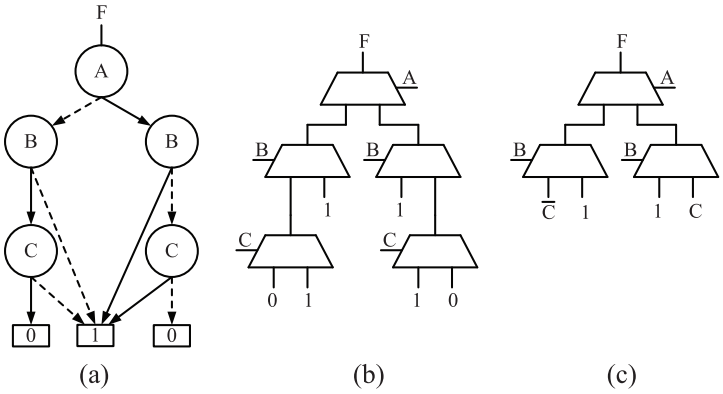


Fig. 5. The BDD method for false-positive-free GLIFT logic generation.

By comparison, GLIFT logic generated using the constructive method includes additional terms (those in the box) indicating nonexistent flows of information. The BDD method eliminates such false positives, resulting in tracking logic that precisely measures all actual information flows.

In the BDD method, the only source of false positives is the single-variable switch in the select line of MUX-2. Now that such false positives are eliminated by instantiating precise GLIFT logic for MUX-2 given in (32), the BDD method is false positive free. In addition, GLIFT logic can only contain false positives and never indicates any false negatives [Hu et al. 2011]. Thus, the BDD method will not lead to removal of any actual information flows.

5.3. Complexity Analysis

According to Section 2.5, the complexity of the general GLIFT problem is $O((2m)^n)$, where m and n are the numbers of security classes and inputs, respectively. The complexity of the naive label propagation rule-set enumeration method can reach $O((2m)^n)$ since it needs to enumerate an n-dimensional label propagation rule set with $2m$ alternative security classes in each dimension. The complexity of the constructive method varies from the GLIFT library it maintains. When a minimum GLIFT library consisting of two-input Boolean gates is maintained, the complexity of the constructive method will reach its lower bound $O((2m)^2 + g)$, where g is the number of primitive gates in the synthesized netlist. The complexity of calculating all prime implicants or constructing a BDD is on the order of $O(2^n)$ while deriving tracking logic for bound operators has complexity of $O(m^2)$ [Palopoli et al. 1999; Lin and Devadas 1995]. Therefore, the precise GLIFT logic generation problem has complexity of $O(m^2 + 2^n + g)$.

The BDD method provides a more efficient solution to false-positive-free GLIFT logic generation (than calculating all prime implicants) since BDD maintenance is quite inline with modern logic synthesis techniques. This will allow precise measurement of actual information flows in large digital circuits. However, precisely accounting for all information flows may lead to high design overheads in area and performance, which will be shown in the experimental results section.

6. GLIFT FOR ENFORCING MULTILEVEL SECURITY

In this section, we will illustrate how GLIFT can be used to enforce MLS. In addition, various design optimization considerations will be addressed.

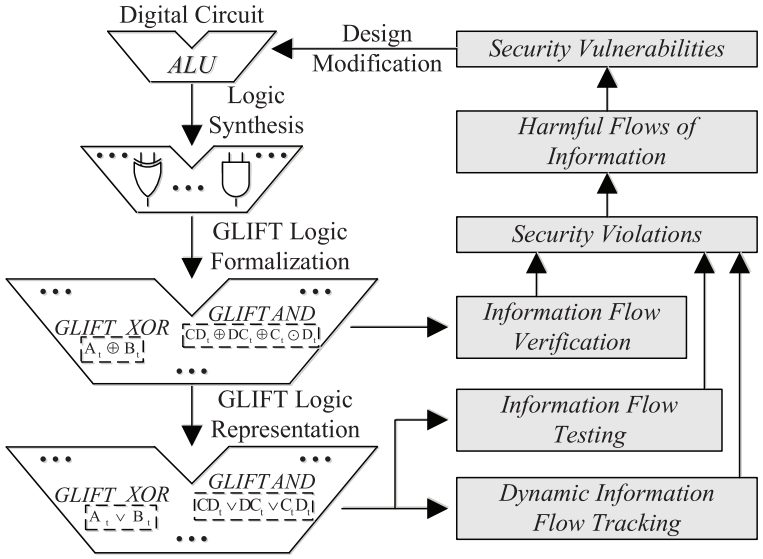


Fig. 6. GLIFT for static information flow testing/verification or dynamic information flow tracking.

6.1. Application Scenarios of GLIFT

There are two application scenarios of GLIFT, namely, *static* or *dynamic*.

Figure 6 illustrates the design flow of GLIFT, either for static information flow testing/verification or dynamic information flow tracking. The differences between the static and dynamic application scenarios lie in that static testing/verification will prevent the additional area and performance overheads introduced by GLIFT logic but require extensive testing efforts. By comparison, dynamic IFT will allow capture of more realistic runtime behaviors at significant resource and performance costs.

In both the static and dynamic application scenarios, the designer is responsible for specifying the security classifications for the inputs according to security specification of the design. To assign the classification levels, he needs to classify the inputs according to their source and security property, for example, inputs taking information from the open environment should be marked as untrusted, and in a cryptographic core, the message, secret key, and control signals can have different security levels. After assigning classification levels to the inputs, GLIFT logic will perform security label propagation and determine the classification levels of the internal nets and outputs.

6.2. GLIFT for Static Information Flow Testing/Verification

In a static testing/verification scenario, the digital circuit under test is first synthesized to a gate-level netlist using logic synthesis tools such as *Synopsys Design Compiler*. Given a security lattice, GLIFT logic for the target digital circuit can be generated using the constructive or BDD method. In the static verification scenario, GLIFT logic is in formal representation, described with security label vectors and bound operators, whereas in the static testing scenario, GLIFT logic needs to be further represented in Boolean functions. Subsequently, the designer can perform security partitioning across the design and run verification scenarios to check whether there is any security violation. Once a violation is identified, the designer can capture the harmful flows of information that caused the violation. Further, security vulnerabilities can be located by tracking backwards along the propagation path of harmful information flows.

Table II. The Least Upper and Greatest Lower Bound Operations on the Square Security Lattice

.lub	UC	S1	S2	TS
UC	UC	S1	S2	TS
S1	S1	S1	S1/S2	TS
S2	S2	S1/S2	S2	TS
TS	TS	TS	TS	TS

.glb	UC	S1	S2	TS
UC	UC	UC	UC	UC
S1	UC	S1	S1/S2	S1
S2	UC	S1/S2	S2	S2
TS	UC	S1	S2	TS

For a more concrete understanding of the difference between static information flow testing and verification, consider the MUX-2 example. The formal representation of GLIFT logic for MUX-2 given in (28) or (32) can be used for verification depending on the precision requirement. Such GLIFT logic representation is independent of the security lattice under consideration, which is applied during the final verification stage. For information flow testing, GLIFT logic should be further represented in Boolean functions. Such a Boolean function varies from security lattices and encoding schemes that need to be specified during the early Boolean function representation stage. As an example, (32) needs to be implemented as (35) under the two-level linear security lattice LOW $\sqsubset$ HIGH, where G_t, A_t, B_t, and S_t are security label bits of G, A, B, and S, respectively. When another security lattice is considered, the resulting Boolean function would be completely different.

$$G_t = SA_t \vee \overline{S}B_t \vee \overline{A}BS_t \vee A\overline{B}S_t \vee A_tS_t \vee B_tS_t \tag{35}$$

Although static testing/verification can prevent the area and performance overheads of additional GLIFT logic, a major limitation of static analysis lies in coverage. The size of state space of a design grows exponentially to its number of inputs. As a result, either extremely long testing/verification time is needed to guarantee full state space coverage or a full coverage can hardly be achieved. Thus, in security-critical applications where information flow control should be strictly enforced, we need to physically deploy GLIFT logic for real-time monitoring of all flows of information.

6.3. GLIFT for Dynamic IFT

In a dynamic application scenario, GLIFT logic needs to be represented in Boolean functions and physically implemented with the original design. A first step in this process is to derive Boolean tracking logic for the bound operators.

6.3.1. Deriving GLIFT Logic for Bound Operators. The tracking logic for bound operators can be described with truth tables that are an implementation of the partial order on the security lattice. Specifically, for a security lattice with m security classes, there would be m^2 entries in the truth tables. As an example, consider the square security lattice as shown in Figure 4(d). Let UC, S1, S2, and TS denote the Unclassified, Secret1, Secret2, and Top Secret security classes, respectively. The least upper and greatest lower bound operations on the square security lattice can be described in Table II.

From Table II, the least upper and greatest lower bound operations on incomparable security classes can lead to nondeterministic output labels (e.g., S1 $\odot$ S2 = S1/S2). In such cases, the designer is responsible for specifying the output label without violating the information flow policy. In this example, either S1 or S2 is a safe label. Since enumerating these truth tables has polynomial complexity, the tracking logic for bound operators can always be derived in polynomial time under a given security lattice.

As a special case, we can reduce Table II for a two-level security lattice that has only two security classes Unclassified and Secret1. For binary implementation, Unclassified and Secret1 can be encoded as "0" and "1", respectively. In this case, the least upper and greatest lower bound operations can be described with a logical

OR and a logical AND array correspondingly. Subsequently, GLIFT logic formalized in Section 4 will reduce to the special case for a two-level security lattice as formalized in Hu et al. [2011]. Such reducibility implies compatibility of our results with preliminary work.

6.3.2. Generating Optimized GLIFT Logic. While deriving Boolean GLIFT logic for physical implementation, the security classes need to be encoded in binary bits. To derive optimized GLIFT logic, we need to choose an encoding scheme that leads to optimization of the most computationally intensive operations, that is, calculating the least upper and greatest lower bounds of security labels. Considering the square lattice shown in Figure 4(d), encoding `Unclassified`, `Secret1`, `Secret2`, and `Top Secret` as "00", "01/10", "10/01", and "11", respectively, will lead to optimized (not necessarily optimal) implementation results since computing the least upper and greatest lower bounds reduces to simple logical AND and OR operations to the greatest extent.

In addition, we can denote unused encoding states as *don't-care* conditions for further optimization. For a security lattice with m security classes, we need at least $w = \lceil log_2 m \rceil$ Boolean bits to encode the security classes. There are $2^w - m$ unused encoding states that can be used as *don't-care* conditions. Denoting such *don't-care* conditions to logic synthesis tools will lead to optimized implementation results.

As an example, consider the GLIFT logic for AND-2 under the three-level linear confidentiality lattice shown in Figure 4(b). According to our analysis, encoding security classes `Unclassified`, `Confidential`, and `Secret` as "00", "01/10", and "11" will lead to optimized implementation results without accounting for the *don't-care* conditions. Considering the case when `Confidential` is encoded as "01", the GLIFT logic can be derived from (17) as (36), where $O_t = (o_t^1, o_t^0)$, $A_t = (a_t^1, a_t^0)$, and $B_t = (b_t^1, b_t^0)$.

$$\begin{aligned} o_t^1 &= A\overline{a_t^1} b_t^1 b_t^0 \vee B a_t^1 a_t^0 \overline{b_t^1} \vee a_t^1 a_t^0 b_t^1 b_t^0 \\ o_t^0 &= A\overline{a_t^1} b_t^0 \vee B a_t^0 \overline{b_t^1} \vee a_t^0 b_t^0 \end{aligned} \tag{36}$$

Under the chosen encoding scheme, "10" is an unused encoding state. When such *don't-care* conditions are denoted to logic synthesis tools, we can obtain the optimized GLIFT circuit as shown in (37).

$$\begin{aligned} o_t^1 &= A b_t^1 \vee B a_t^1 \vee a_t^1 b_t^1 \\ o_t^0 &= A b_t^0 \vee B a_t^0 \vee a_t^0 b_t^0 \end{aligned} \tag{37}$$

By comparison of (36) and (37), denoting *don't-care* conditions can lead to significant optimization effects. It is important to point out that such optimization will not cause any security violation since the input pattern "10" will never be observed at the security label inputs.

7. EXPERIMENTAL RESULTS

In this section, we first perform a comparison of GLIFT logic generated by different methods in terms of area, delay, and precision using IWLS benchmarks. We then demonstrate how GLIFT can be used for enforcing information flow security through static testing/verification. Finally, we present area and performance analysis of GLIFT logic for Trust-Hub [Baumgarten et al. 2011] and IWLS [2005] benchmarks to show the design overheads that should be expected when expanding GLIFT for MLS.

7.1. A Comparison of GLIFT Logic Generation Methods

We use the IWLS benchmark *x2*, a moderate design with limited number of I/Os, for precision analysis. GLIFT logic for this benchmark under the four-level confidentiality security lattice shown in Figure 4(c) is generated using both the constructive and BDD

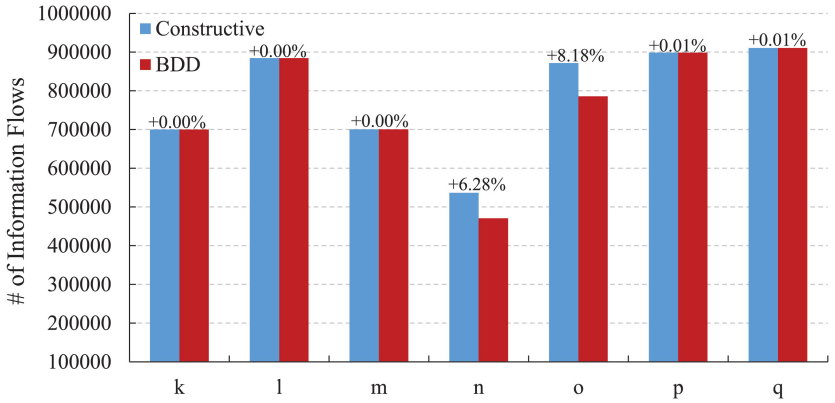


Fig. 7. The number of information flows in GLIFT logic generated using different methods. k through q are the outputs of the benchmark.

methods. The `Unclassified`, `Confidential`, `Secret`, and `Top Secret` security classes are encoded as "00", "01", "10", and "11", respectively. The GLIFT circuits are simulated under a total number of 2^{20} random test vectors produced by a Linear Feedback Shift Register (LFSR) to see how often each output takes a security label other than `Unclassified`, which reflects the number of information flows. Figure 7 shows the experimental results, where k through q represent the outputs of the benchmark. The percentage data reflect the ratio of false positives in GLIFT logic generated by the constructive method.

Taking the output n as an example, the GLIFT logic created using the constructive method contains 6.28% false positives. These false positives correspond to $2^{20} * 0.0628 = 65850$ nonexistent information flows, indicating that secret information leaks to unclassified domains when actually it does not. Such false positives can lead to additional verification time or conservative system behaviors by producing false security alters and thus should be reduced to the greatest extent whenever possible.

We then generate GLIFT logic for several IWLS benchmarks under the same security lattice and encoding scheme using both the constructive and BDD methods for a comparison of area and performance. In the constructive method, a minimum GLIFT library consisting of the tracking logic for AND-2, OR-2, and INV is maintained for simplicity. Maintaining such a minimum library will not hinder the validity of our experimental results since the GLIFT logic needs to be resynthesized and mapped to the Synopsys SAED 90nm standard cell library [Synopsys 2007] for area, delay, and power reports. The results are shown in Table III. The table also shows the total number of operators (least upper/greatest lower bound and dot product) in the formal representation of GLIFT logic. Such a number reflects the complexity of the GLIFT logic independent of the security lattice.

From Table III, it can be seen that GLIFT logic generated using the constructive and BDD methods may see significant differences in the total number of operators, area, and performance. For most benchmarks, the constructive method will result in smaller area, delay, and power consumption. However, GLIFT logic generated by the BDD method may sometimes see small delay due to logic-level reductions during the BDD construction phase. On average, GLIFT logic created using the BDD method reports 4.51× in total more number of operators, 3.65× in area, 1.74× in delay, and 3.43× in power, respectively, than those generated by the constructive method.

In practice, the designer needs to balance between precision requirements and design overheads and decide what GLIFT logic generation method should be used. In

Table III. Total Number of Operators, Area, Delay and Power of GLIFT Logic (including the original design) under the Four-Level Linear Lattice Generated by Different Methods

	# Operators		Area (um^2)		Delay (ns)		Power (mW)	
Benchmark	Cons.	BDD	Cons.	BDD	Cons.	BDD	Cons.	BDD
alu2	1884	1881	23047	22863	2.43	2.01	5.10	5.05
alu4	3744	8877	43126	102751	3.59	2.97	9.47	22.0
apex6	3756	10351	46946	110684	1.47	3.49	8.91	21.5
x3	3552	9592	46381	108552	1.26	3.48	8.86	22.0
x4	1710	10043	20968	91776	1.04	3.00	4.28	18.7
i5	792	6677	12069	40755	1.84	2.49	2.38	7.37
i6	2310	3058	26529	25765	1.52	1.10	6.51	5.70
i7	3054	3388	35695	48714	1.64	1.25	8.89	9.45
i8	4770	23815	59438	380678	1.86	2.70	13.6	77.9
i9	3036	21769	41174	312563	1.86	2.78	11.1	66.2
frg2	4038	36278	54854	279606	2.22	4.87	11.0	57.1
pair	11766	86757	102704	669799	2.25	7.13	21.3	143
N. Average	1.00	4.51	1.00	3.65	1.00	1.74	1.00	3.43

safety-critical systems where false positives are strictly not allowed, the BDD method needs to be used, whereas in systems where a certain amount of false positives can be tolerated, the constructive method would be sufficient and much more cost effective.

7.2. Static Testing/Verification Analysis

We use the Trust-Hub [Baumgarten et al. 2011] benchmark *BasicRSA-T400*, that implements the 32-bit RSA cryptographic algorithm, for static testing/verification. We show how GLIFT can capture secret key leakage through a hidden timing channel.

In the experiment, we target the same security lattice and use an identical encoding scheme as in Section 7.1. We divide the 32-bit secret key into four bytes and label these bytes from MSB to LSB as type `Top Secret` (“11”), `Secret` (“10”), `Confidential` (“01”), and `Unclassified` (“00”), respectively. Therefore, the security label of the entire secret key is *key_t = 64'hFFFF_AAAA_5555_0000* (each key bit has a 2-bit label). We focus on the secret key in our test and thus mark all the remaining signals as type `Unclassified`. The GLIFT logic for the benchmark is created using the constructive method, represented in Boolean function and simulated under ModelSim to capture secret key leakage. The simulation result is shown in Figure 8.

From Figure 8, note that GLIFT logic indicates that the secret key flows to the *cipher* and *rdy* signals since both *cipher_t* and *rdy_t* take labels other than `Unclassified`. We concentrate on the *rdy* signal in our analysis since the RSA cryptographic algorithm prevents secret key leakage via cipher text. It is shown that *rdy_t* changes from `Unclassified` to `Confidential` and eventually to `Top Secret`, indicating the algorithm implementation processes from LSB to MSB. However, the *rdy* signal always remains constantly logical “0” until the encryption completes, regardless of the value of the secret key. Thus, the secret key does not affect the *rdy* signal in a way that determines whether *rdy* can take a logical “1” value. Instead, it has an influence on *rdy* by determining when it actually takes a logical “1”. In other words, the secret key leaks to *rdy* via timing-related flow. Such timing flow is caused by the timing difference between different algorithm branches selected by the secret key bits. Further, the secret key can be retrieved through statistical analysis of the algorithm processing time measured at the *rdy* signal [Kocher 1996]. Similarly, more test scenarios can be run by assigning security labels to the signals and observing if there is any harmful flow of information.

From the experiment, we see GLIFT can be used to capture harmful timing flows that indicate the existence of hidden timing channels. By employing a multilevel security

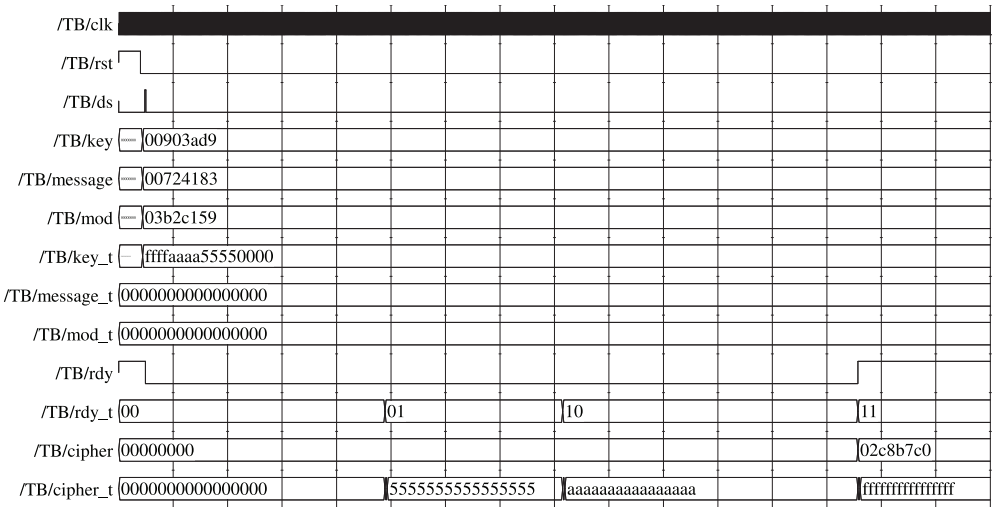


Fig. 8. GLIFT logic captures key leakage during static information flow testing.

lattice, it allows finer classification of data objects and a better understanding of the leakage process. We refer the interested reader to other works that provide more detail on GLIFT for detecting timing channels [Oberg et al. 2013a; 2014], which is out of the scope of this work.

7.3. Design Complexity of GLIFT for Dynamical IFT

We also generate GLIFT logic for several Trust-Hub [Baumgarten et al. 2011] and IWLS [2005] benchmarks under the two-to-four-level linear and square security lattices in Figure 4. The tracking logic (including the original design) is created using the constructive method, synthesized using Synopsys Design Compiler, and targeted to the Synopsys SAED 90nm standard cell library [Synopsys 2007] for area and delay reports, as shown in Table IV. The table also shows the total number of operators (least upper bound, greatest lower bound, and dot product) in the formal representation of GLIFT logic. Such numbers remain constant under different security lattices.

Observe from Table IV that GLIFT logic typically reports larger area and delay as the security lattice grows more complex. Row "*N. Average*" shows the average area and delay normalized to those under the two-level security lattice, reflecting the design overheads that should be expected when expanding GLIFT to multilevel security lattices. It should be noticed that tracking logic under the three-level linear lattice consumes relatively smaller area and delay than that for the four-level one. This is because we took the don't-care input set into account and denoted these don't-cares conditions to the logic synthesis tool.

From the experimental results, we see that expanding GLIFT to multilevel security lattices will result in considerable area and performance overheads. However, realistic systems usually require MLS policies that need to be modeled using multilevel security lattices, thus the two-level GLIFT method should be expanded to meet such requirements. Security is a pressing problem in high-assurance systems that are being confronted with rapidly evolving cyber threats. Such design overheads should definitely be tolerated since a single failure resulting from security vulnerabilities will render critical infrastructures useless and cause tremendous losses. In real applications, there are usually partitions among security domains within a design. Only security-critical portions of the design need to be augmented with GLIFT logic for dynamic IFT.

Table IV. Total Number of Operators, Area and Delay of GLIFT Logic (including the original design) under the Two-to-Four-Level linear and Square Lattices

		Area (um^2)				Delay (ns)			
Benchmark	# Oper.	2-lev	3-lev	4-lev	Square	2-lev	3-lev	4-lev	Square
s1423	3672	20096	49208	62308	68435	2.18	2.52	2.62	4.01
s5378	7392	39527	99659	125364	142428	1.54	1.96	2.03	2.49
s9234	5742	31964	79796	100979	113922	0.48	0.64	0.70	1.17
s13207	12618	52589	137231	162122	206494	1.07	1.89	2.11	2.79
s15850	19680	35581	97020	116675	126359	0.49	0.76	0.70	0.92
s35932	71136	458857	1160821	1511884	1705441	0.57	1.29	1.10	1.38
s38417	98310	569453	1511936	2136536	2310636	1.12	1.28	1.65	1.83
s38584	61896	340235	897934	1208826	1322753	1.35	2.81	1.88	2.02
DES	17712	45845	154853	204085	244764	2.18	2.76	2.62	3.03
BasicRSA	30204	129282	354764	432836	508598	0.14	0.33	0.48	0.34
b19	688050	2326801	6184741	7848774	8266121	6.66	8.73	8.61	12.0
RS232	1740	11459	27262	32285	35965	0.62	0.81	1.01	1.17
wb_conmax	225294	992830	3514102	4403386	4614289	2.61	3.38	4.16	4.62
PIC16F84	13416	64282	184123	223647	249298	0.50	0.74	0.71	0.86
MCU8051	56148	237109	692763	874074	991815	11.2	15.3	17.0	25.3
N. Average	–	1.00	2.74	3.47	3.89	1.00	1.53	1.62	1.95

In addition, GLIFT can also be used for static information flow security testing or verification, which does not require physical implementation of the additional tracking logic. These options will reduce or eliminate the area and performance overheads.

8. CONCLUSION

GLIFT provides an effective approach for detecting security vulnerabilities, including hard-to-detect timing channels that are inherent in the underlying hardware but invisible to higher levels of abstraction. This article provides a general way to expand the GLIFT method to target multilevel security lattices, formalizing tracking logic for primitive gates, presenting precise GLIFT logic generation methods for large digital circuits, and demonstrating various application scenarios of GLIFT, which provides a possibility for proving MLS in high-assurance systems from the ground up. It also reveals what sort of area and performance overheads should be expected if MLS is required at such a low level of abstraction. However, the present state-of-the-art of GLIFT cannot handle attacks that require statistical analysis or account for information flows caused by physical phenomena, such as power dynamics or electromagnetic radiation, that do not appear at the logical level. In our future work, we will investigate various design optimization issues such as different mapping techniques, optimized encoding schemes, and formal verification methods.

REFERENCES

Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2006. Predicting secret keys via branch prediction. In *Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'07)*. Springer, 225–242.

Alex Baumgarten, Michael Steffen, Matthew Clausman, and Joseph Zambreno. 2011. A case study in hardware trojan design and implementation. *Int. J. Inf. Secur.* 10, 1, 1–14.

D. Elliott Bell and Leonard J. LaPadula. 1973. Secure computer systems: Mathematical foundations. http://www.albany.edu/acc/courses/ia/classics/belllapadula1.pdf.

Daniel J. Bernstein. 2005. Cache-timing attacks on aes. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive experimental

analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, 6.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM Press, New York, 482–493.

Dorothy E. R. Denning. 1976. A lattice model of secure information flow. *Comm. ACM* 19, 5, 236–243.

Dorothy E. R. Denning. 1982. *Cryptography and Data Security*. Addison Wesley Longman, Boston, MA.

Edward B. Eichelberger. 1965. Hazard detection in combinational and sequential switching circuits. *IBM J. Res. Develop.* 9, 2, 90–99.

Joseph A. Goguen and Jose Meseguer. 1982. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'82)*. 11–20.

Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. 2008. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'08)*. 129–142.

Wei Hu, Jason K. Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2011. Theoretical fundamentals of gate level information flow tracking. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 30, 8, 1128–1140.

Wei Hu, Jason K. Oberg, Dejun Mu, and Ryan Kastner. 2013. Expanding gate level information flow tracking for multi-level security. *IEEE Embedd. Syst. Lett.* 5, 2, 25–28.

IWLS. 2005. IWLS benchmarks ver. 3.0. http://iwls.org/iwls2005/benchmarks.html.

Ryan Kastner, Jason K. Oberg, Wei Hu, and Ali Irturk. 2011. Enforcing information flow guarantees in reconfigurable systems with mix-trusted ip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'11)*.

Paul C. Kocher. 1996. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96)*. Springer, 104–113.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.* 41, 6, 321–334.

Chunxiao Li, Anand Raghunathan, and Niraj K. Jha. 2011. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *Proceedings of the 13th IEEE International Conference on e-Health Networking Applications and Services (Healthcom'11)*. 150–156.

Bill Lin and Srinivas Devadas. 1995. Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 14, 8, 974–985.

Anil K. Maini. 2007. *Digital Electronics: Principles, Devices and Applications*. John Wiley and Sons.

Yilin Mo, Tiffany Hyun-Jin Kim, Kenneth Brancik, Dona Dickinson, Heejo Lee, Adrian Perrig, and Bruno Sinopoli. 2012. Cyber-physical security of a smart grid infrastructure. *Proc. IEEE* 100, 1, 195–209.

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*.

Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging gate-level properties to identify hardware timing channels. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* 33, 9, 1288–1301.

Jason K. Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2011. Information flow isolation in i2c and usb. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. 254–259.

Jason K. Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2013a. A practical testing framework for isolating hardware timing channels. In *Proceedings of the Design, Automation, and Test in Europe Conference (DATE'13)*. 1281–1284.

Jason K. Oberg, Timothy Sherwood, and Ryan Kastner. 2013b. Eliminating timing information flows in a mix-trusted system-on-chip. *IEEE Des. Test* 30, 2, 55–62.

Luigi Palopoli, Fiora Pirri, and Clara Pizzuti. 1999. Algorithms for selective enumeration of prime implicants. *Artif. Intell.* 111, 12, 41–72.

Francois Pottier and Vincent Simonet. 2003. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.* 25, 1, 117–158.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Select. Areas Comm.* 21, 1, 5–19.

Claude E. Shannon. 2001. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Comm. Rev.* 5, 1, 3–55.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.* 38, 5, 85–96.

Synopsys. 2007. SAED edk90 core - 90nm digital standard cell library. http://www.synopsys.com/community/universityprogram/pages/library.aspx.

Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. 2009a. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09).* ACM Press, New York, 493–504.

Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009b. Complete information flow tracking from the gates up. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09).* ACM Press, New York, 109–120.

Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *Proceeding of the 38th Annual International Symposium on Computer Architecture (ISCA'11).* ACM Press, New York, 189–200.

Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazieres. 2007. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.* 25, 4.

Claire Vishik, Ruby B. Lee, and Fred Chong. 2012. Building technologies that help cyber-defense: Hardware-enabled trust. In *Securing Electronic Business Processes: Highlights of the Information Security Solutions Europe Conference,* H. Reimer, N. Pohlmann, and W. Schneider Eds., Springer, 316–325.

Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *J. Comput. Secur.* 4, 2–3, 167–187.

Received September 2013; revised June 2014; accepted July 2014