Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking

Wei Hu[†], Andrew Becker[‡], Armita Ardeshiricham[†], Yu Tai[§], Paolo Ienne[‡], Dejun Mu[§] and Ryan Kastner[†]
[†]University of California, San Diego, La Jolla, CA 92093
[‡]École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
[§]Northwestern Polytechnical University, Xi'an 710072, Shaanxi, China
{weh040, aardeshi, kastner}@ucsd.edu; {andrew.becker, paolo.ienne}@epfl.ch;
taiyu@mail.nwpu.edu.cn, mudejun@nwpu.edu.cn

ABSTRACT

Secure hardware design is a challenging task that goes far beyond ensuring functional correctness. Important design properties such as non-interference cannot be verified on functional circuit models due to the lack of essential information (e.g., sensitivity level) for reasoning about security. Hardware information flow tracking (IFT) techniques associate data objects in the hardware design with sensitivity labels for modeling security-related behaviors. They allow the designer to test and verify security properties related to confidentiality, integrity, and logical side channels. However, precisely accounting for each bit of information flow at the hardware level can be expensive. In this work, we focus on the precision of the IFT logic. The key idea is to selectively introduce only one sided errors (false positives); these provide a conservative and safe information flow response while reducing the complexity of the security logic. We investigate the effect of logic synthesis on the quality and complexity of hardware IFT and reveal how different logic synthesis optimizations affect the amount of false positives and design overheads of IFT logic. We propose novel techniques to further simplify the IFT logic while adding no, or only a minimum number of, false positives. Additionally, we provide a solution to quantitatively introduce false positives in order to accelerate information flow security verification. Experimental results using *IWLS* benchmarks show that our method can reduce complexity of GLIFT by 14.47% while adding 0.20% of false positives on average. By quantitatively introducing false positives, we can achieve up to a 55.72% speedup in verification time.

1. INTRODUCTION

Digital hardware lies at the core of modern computing systems; it computes and manages every single piece of data in a computer system. However, the sheer size and complexity of the hardware design make it almost inevitable that it will have security flaws, and there have been a number of reports about hardware security issues in critical devices responsible for protecting personal privacy, controlling high-assurance systems, and operating military weapons [1, 4, 27]. This is largely due to the fact that the traditional hardware design flow and tool-chain provide limited support for reasoning about security. Hardware designers are in desperate need for tools that enable them to assess and understand potential security vulnerabilities in their design.

A hardware design is typically described using a functional circuit model, e.g., Boolean functions and gates. Such models do not carry the additional information required to understand its security. As a result, important security properties such as non-interference [7] cannot be verified on hardware designs using existing tools. A classic technique called information flow tracking (IFT) is frequently used to reason about the security of a system [5, 26]. It is highly effective in modeling and enforcing security properties related to confidentiality and integrity.

IFT has been widely deployed across the entire system stack, including programing language/compiler [22], operating system [13], and instruction set architecture [23]. Gate level information flow tracking (GLIFT) enforces information flow security at the hardware level [25]. It captures different types of logical flows (data flows, control flows and timing flows) in a unified manner. GLIFT has been employed to prove isolation between IP cores of different trust in SoC systems [12], detect and eliminate timing channels in shared bus architectures such as *I2C* and *USB* [20], and craft verifiably information flow secure architectures [24].

However, creating a precise GLIFT model that can accurately measure each bit of information flow has been proved to be an NP-complete problem [10]. A fast and common method for deriving the GLIFT circuit (also called GLIFT logic) uses a constructive technique [10]. This allows for polynomial time generation but it also introduces a certain amount of non-existent flows of information. These additional flows represent *false positives.* They may state that a flow occurs when it actually does not. It is a safe and conservative approach, i.e., it will never say a security property holds when it in fact is violated. But it may falsely state that the hardware violates the security property when actually it does not.

Imprecise GLIFT logic is typically smaller in size since the imprecise GLIFT model for logic primitives tends to be simpler. This can reduce security verification time be-

ICCAD '16, November 07-10, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4466-1/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2966986.2967046

cause there is evidence that smaller circuits are easier to verify [19]. Additionally, the imprecise GLIFT logic may catch an information flow violation earlier. To wit, we show later in this paper that the verification of a security property using imprecise GLIFT logic can be proven (i.e., the verification solver will complete in a reasonable amount of time) while the same property on the same hardware design will not complete using precise GLIFT logic.

Imprecise GLIFT logic can be generated by adding false positives. Such false positives indicate flows of information when actually there is none. They can be introduced during different phases of the design flow. As an example, logic synthesis can affect the flow of information by changing the connectivity of signals. In addition, one can create different GLIFT models for logic primitives in the technology library and perform mapping selectively.

In this paper, we investigate various tradeoffs in quality and complexity of hardware IFT. We focus on the effects of logic synthesis optimizations and different mapping strategies on the precision and complexity of GLIFT logic. Specifically, this paper makes the following contributions:

- Revealing the effect of logic synthesis techniques on the precision and complexity of hardware IFT;
- Proposing techniques to simplify GLIFT logic while adding no or a minimum amount of false positives;
- Providing a solution to quantitatively introduce false positives to accelerate information flow security verification and localize security vulnerability.

The reminder of this paper is organized as follows. Section 2 covers how GLIFT measures hardware information flows and the generation of GLIFT logic. In Section 3, we discuss how logic synthesis can affect the precision and complexity of GLIFT logic. We provide methods that allow tradeoff precision and design overheads through library mapping in Section 4. Section 5 presents experimental results using *IWLS* benchmarks. We briefly review related work in Section 6 and conclude the paper in Section 7.

2. BACKGROUND

2.1 Gate Level Information Flow Tracking

Gate level information flow tracking (GLIFT) is built upon the notion that information flows from a signal A to B *if and only if* the value of A has an influence on B. GLIFT associates a label (sometimes called taint) with each data bit in the hardware design to mark the information assets one would like to track. The security labels for internal signals and output ports are calculated and updated according to the type of logic operation performed. By checking the security labels of the signals, one can determine if the design violates different security properties. Examples of such properties include: 1) confidential data should never leak to an unclassified domain, and 2) untrusted data should never be allowed to overwrite a trusted memory location. GLIFT captures explicit flows (also called data flows), implicit flows (also known as control flows), and timing flows (information flow through a timing side channel) in a unified manner since they all appear in a similar form at the gate level.

2.2 GLIFT Precision and Complexity

It is possible to generate different variants of GLIFT logic. The precision and performance depends on the granularity of security label and label propagation policy used. For a better understanding, consider a two-input AND (AND-2) gate, whose Boolean function can be described as $O = A \cdot B$. We use A_t, B_t and O_t to denote the security labels of the inputs and output, respectively. To make it easier to understand, we restrict to single bit variables and labels in the following discussion. However, it is straightforward to generalize to multi-bit variables.

Let A_t be `1` when A is `Confidential` and `0` when A is `Unclassified` (similar for B_t and C_t). GLIFT can use an imprecise policy shown in Equation (1) to determine O_t.

$$O_t = A_t + B_t \tag{1}$$

This is secure since it precisely captures all possible confidential information flows from either A or B to the output. However, it can be imprecise since it may over estimate actual flows of information. As an example, assume A is `Unclassified` logical `0`, and B is carrying `Confidential` information, i.e., $B_t = 1$. In this case, O will be dominated by A; the other confidential input has no effect on the output. In other words, there is no flow of confidential information. However, Equation (1) would conservatively state that $O_t = 1$, i.e., O is `Confidential`.

GLIFT provides a more precise approach to calculate the security label for the output. It determines a flow of information from an input to the output when the value of the input has an influence on that of the output. Consider again the AND-2 example. When both inputs are `Confidential`, the output will undoubtedly be `Confidential`. This introduces the term $A_t \cdot B_t$ in the GLIFT logic. When A is `Confidential`, the condition for it to be observable at the output should be $B = 1$. This yields the term $B \cdot A_t$ in the tracking logic. Similarly, $A \cdot B_t$ should also be added to track the `Confidential` information flowing from B to the output when $A = 1$. From the above analysis, the GLIFT logic for AND-2 can be formalized as:

$$O_t = A \cdot B_t + B \cdot A_t + A_t \cdot B_t \tag{2}$$

Let *Secret* be a 32-bit `Confidential` value. After performing the operation in Equation (3), the imprecise model shown in Equation (1) will mark the entire *Public* signal as `Confidential`, indicating there are 32 bits of information flowing from *Secret* to *Public* (or simply there is a flow).

$$Public = Secret \ \cdot \ \text{0x03} \tag{3}$$

Using Equation (2), the `Unclassified 0` bits in the second operand will dominate the security labels of corresponding output bits. Thus, only the lowest two bits of `Public` will be labeled as `Confidential`, indicating there are two bits of `Confidential` information flowing to `Public`.

However, this precision comes at the cost more complex GLIFT logic. As an example, the precise GLIFT logic for AND-2 shown in Equation (2) is more complicated than the imprecise one shown in Equation (1). This may increase the time for verification of security properties.

2.3 GLIFT Logic Generation

A fundamental problem involved in GLIFT logic generation is to determine if a tainted input can flow to some (set of) signal(s). This reduces to several known hard problems such as Boolean satisfiability and controllability [6]. To generate GLIFT logic in acceptable amount of time, one typically resorts to the *constructive method* as shown in Fig. 1.

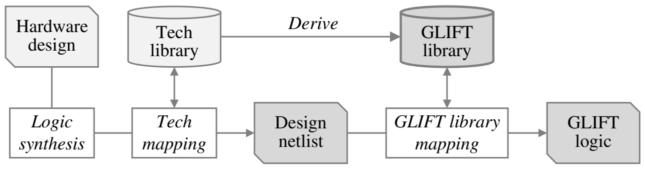

Figure 1: The constructive method for GLIFT logic generation.

This constructive method first constructs a GLIFT logic library (GLIFT library) consisting of the GLIFT logic for each primitive gate in a technology library. The hardware design is then synthesized and mapped to that technology library. Each logic primitive in the mapped design netlist is instantiated with GLIFT logic from the GLIFT library discretely. In this way, the complexity of the GLIFT logic generation problem is dominated by logic synthesis (and technology mapping) since mapping logic primitives in the design netlist to the GLIFT library can be completed in linear time. However, the GLIFT logic generated in this constructive manner can be imprecise, i.e., it may contain false positives that indicate non-existent flows of information [10].

Logic synthesis and the target technology library have a significant effect on the generated GLIFT logic. Different logic synthesis strategies can lead to significantly different representations of a given hardware design, e.g., collapsed PLA or BDD tree. The technology library determines the required GLIFT library (there should be GLIFT logic corresponding to each gate in the technology library). Changes in the logic synthesis strategy and/or the technology library can lead to variable GLIFT logic. The differences primarily lie in two aspects: precision and complexity. We measure precision using the amount of false positives while quantifying complexity with parameters such as area, delay, and verification time. In the following, we attempt to reveal the effect of logic synthesis on the precision and complexity of GLIFT logic in Section 3. Then we provide various approaches to trade off precision and design overheads through GLIFT library mapping in Section 4.

3. LOGIC SYNTHESIS OPTIMIZATIONS

Logic synthesis affects the precision and complexity of GLIFT primarily due to the fact that different functionally equivalent logic networks can result in variable GLIFT logic. The difference in the logic network may affect the way in which signal flows through hardware design. Consider a two-input XOR (XOR-2) example with `11` as a don't-care input condition. The XOR can be simplified to a two-input OR (OR-2) gate, which is relatively smaller. The GLIFT logic for XOR-2 is the same as Equation (1) while the GLIFT for OR-2 is shown in Equation (4). In this case, simplifying the design will increase the complexity of the GLIFT logic.

$$O_t = \overline{A} \cdot B_t + \overline{B} \cdot A_t + A_t \cdot B_t \tag{4}$$

Logic synthesis changes the precision of the GLIFT logic in two ways. On one hand, logic synthesis can reduce design redundancy. Consider the absorption law [2] shown in Equation (5) that is frequently used for logic optimization.

$$A + \overline{A} \cdot B = A + B \tag{5}$$

The GLIFT logic for the left-hand side of Equation (5) is:

$$O_t = A_t + \overline{A} \cdot B_t \tag{6}$$

By comparison, the GLIFT logic for the right-hand side is shown in Equation (4). Comparing Equations (4) and (6), we see that logic synthesis can possibly make the GLIFT logic more precise. This is because logic synthesis eliminates the reconvergent fanout at A.

On the other hand, logic synthesis tends to increase resource sharing in order to reduce area consumption. This will typically make the resulting GLIFT logic less precise because the resource sharing will lead to additional reconvergent fanout regions. Generally, precision and complexity are contradictory design choices. We present experimental results to show such tradeoffs in Section 5.1.

4. GLIFT LIBRARY MAPPING

4.1 Creating Simplified GLIFT Library

The GLIFT library can be created by enumerating truth tables for primitive gates since these logic primitives usually have a limited number of inputs. Figure 2 shows the Karnaugh maps of GLIFT logic for AND-2 given in Equations (2) and (1). These represent the most precise and imprecise GLIFT logic for AND-2, respectively.

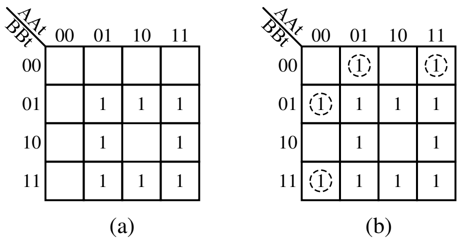

Figure 2: Karnaugh maps of different GLIFT logic for AND-2. (a) Precise GLIFT logic for AND-2 derived from Equation (2). (b) An imprecise version of GLIFT logic for AND-2 corresponding to Equation (1).

Additional GLIFT logic can be derived by gradually flipping the minterms that differ between the two Karnaugh maps. For the four minterms that show a difference, there are a total of $2^4 - 1 = 15$ (excluding the "all flipped" case, which corresponds to the least precise) possible combinations. After testing all these 15 combinations, we derive two additional versions of GLIFT logic for AND-2 that lies between the most and least precise. These are:

$$\begin{aligned} O_t &= B_t + B \cdot A_t \\ O_t &= A_t + A \cdot B_t \end{aligned} \tag{7}$$

Using a similar approach, we can derive additional versions of GLIFT logic for the OR-2. However, this method becomes intractable even for a three-input AND gate (AND-3), which has 28 different minterms between the most and least precise GLIFT logic. This yields a total of $2^{28} - 1$ possible combinations for the simple AND-3. However, the above example reveals how different versions of GLIFT logic can be derived by flipping non-tainted minterms.

We take a more efficient approach to derive simplified GLIFT logic for more complex Boolean gates. We observe that precise GLIFT logic more accurately measures information flows because it takes into account the value of variables in label propagation. Thus, we can gradually relax precision and move towards the least precise by ignoring the values. Figure 3 shows such a solution using AND-3 as an example.

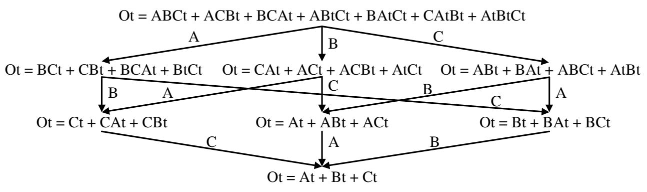
Ot = ABCt + ACBt + BCAt + ABtCt + BAtCt + CAtBt + AtBtCt
A
B
C
Ot = BCt + CBt + BCAt + BtCt
Ot = CAt + ACt + ACBt + AtCt
Ot = ABt + BAt + ABCt + AtBt
B
A
C
B
C
A
Ot = Ct + CAt + CBt
Ot = At + ABt + ACt
Ot = Bt + BAt + BCt
C
A
B
Ot = At + Bt + Ct

Figure 3: Deriving simplified IFT logic for AND-3.

At the top level, we have the most precise GLIFT logic for AND-3, where A, B, C and O are the inputs and output of AND-3; A_t, B_t, C_t and O_t are their security labels, respectively. By setting any of the inputs to don't-care, we can obtain three simplified versions of GLIFT logic as shown at the second level. Now by eliminating an additional remaining input in the simplified versions of GLIFT logic, we can get another three further simplified versions of GLIFT logic shown at the third level. When the last remaining input is eliminated, we reach the least precise (and most simplified) GLIFT logic for AND-3, shown at the bottom level. In this way, we reduce the complexity of the problem to $O(2^n)$, where n is the number of inputs to a Boolean gate. This is tractable since most Boolean gates in the technology library only have a limited number of inputs.

Now that we can create a simplified GLIFT library, we can map logic primitives to each alternative version of its GLIFT logic in order to trade off precision and complexity. In Section 4.2, we provide a new method to simplify GLIFT logic without affecting its precision. We propose another new approach to simplifying GLIFT logic while adding a small number of false positives in Section 4.3. We also present a solution to quantitatively introduce false positives in Section 4.4 to speed up information flow security verification.

4.2 Deriving Simplified GLIFT Logic Without Adding False Positives

Our first approach to GLIFT logic simplification is based on formulating *Quantified Boolean Formulas* (QBFs) to enable GLIFT logic replacement. We check the QBFs' satisfiability (SAT) and recover corresponding valid replacements. In short, this approach asks a QBF-SAT solver to find where and how to replace precise GLIFT logic with less-precise alternatives such that the entire GLIFT circuit is simplified while preserving the overall precision.

We begin by mapping the design netlist to the most precise GLIFT logic. Then, we map each GLIFT element to a multiplexer, selecting from some alternative replacements: less-precise variants. See Fig. 4 for an example of what each multiplexer looks like, where "SH2" is the most precise, "SH3" less precise, and so on. Formally, the variables controlling the multiplexer's select lines compose the existentially-quantified vector $\vec{h}$ in the following formula:

$$\exists \vec{h}\ \forall \vec{i}:\ \text{inst}(\vec{h},\ \vec{i}) \Leftrightarrow \text{spec}(\vec{i}) \tag{8}$$

Equation (8) allows the QBF-SAT solver to find simpler logic, or replacements of certain gates which do not change the observable behavior of the entire GLIFT circuit.

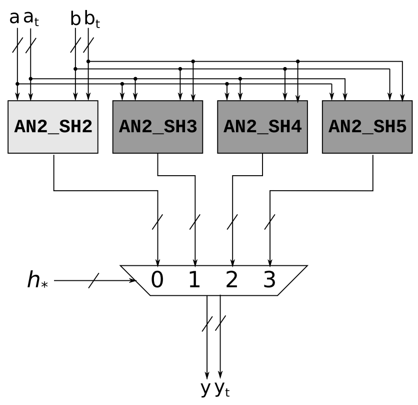

Figure 4: GLIFT logic for AND-2 is mapped to a multiplexer selecting between the precise AN2_SH2 model and the simpler but less-precise alternatives.

To target the search for *better* configurations, we use the idea of Becker et al. [3] to ensure the solver finds a configuration *of a certain quality.* We use the number of GLIFT replacements as a proxy for quality. Because the original logic is selected by an all-`0` select line, we OR-reduce each multiplexer's select line (each such h_* in Fig. 4) to a single bit, and sum them all. This sum represents the number of GLIFT logic replacements with alternatives. The sum is then compared to a threshold value τ, which must be reached to satisfy the QBF. This gives the final QBF, where thresh is the thresholding function:

$$\exists \vec{h}\ \forall \vec{i}:\ (\text{inst}(\vec{h},\ \vec{i}) \Leftrightarrow \text{spec}(\vec{i})) \wedge \text{thresh}(\vec{h},\ \tau) \tag{9}$$

The last step of the process is to run a binary search on τ. Figure 5 shows how it all fits together for one iteration of the binary search. Starting at $\tau = 1$, these QBF problems are formulated, solved, and have their solutions recovered; the difference between the maximum known succeeding value for τ and the minimum known failing value for τ is halved; and the next iteration proceeds with that value. This proceeds until we find a final known maximum succeeding value for τ and a corresponding configuration for $\vec{h}$.

Note that this technique maximizes *the number of replacements*, not the simplicity of entire GLIFT circuit. It is possible that the simplest result does not have the maximum number of replacements: simply, some replacements are better than others. Still, in general, additional replacements almost always improve the result.

Further studies may broaden the scope of considered alternatives. For example, we might consider varying precision levels of both OR-2 and using the GLIFT logic for XOR-2 as replacements for an OR-2, an idea which will be visited thoroughly in Section 4.3.

4.3 Deriving Simplified GLIFT Logic with Minimum Additional False Positives

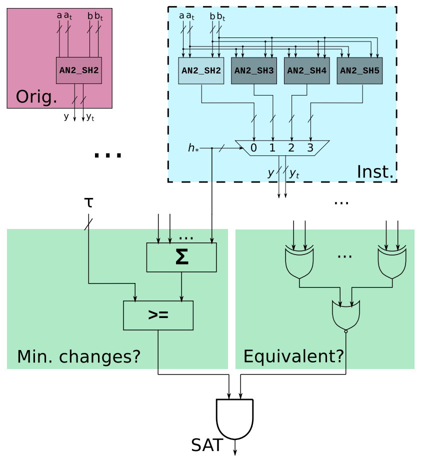

Figure 5: Overview of the instrumented GLIFT logic before being fed to the solver. Note the two conditions: on the left, that $\vec{h}$ corresponds to at least τ replacements; on the right, that with these replacements, the GLIFT logic is functionally identical to the original. Note that the "Inst" logic in the dashed box in the upper-right is not connected to the equivalence check: only GLIFT primary outputs are checked for equivalence.

Our observation is that GLIFT logic generated using the constructive method can be further simplified without significantly affecting precision. This is because the design netlist is created by logic synthesis tools and thus is highly optimized. However, the GLIFT logic is generated by directly mapping the design netlist to the GLIFT library, which does not involve any optimization. Therefore, the GLIFT logic may contain a considerable amount of redundancy. To better understand, consider the two-input multiplexer (MUX-2) example as shown in Fig. 6.

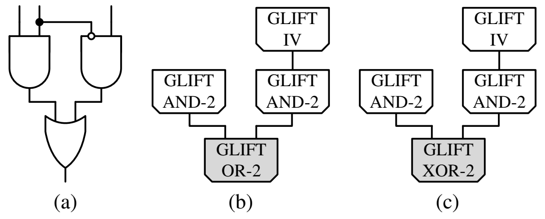

Figure 6: The GLIFT logic for MUX-2 created using the constructive method can be further simplified. (a) Design netlist of MUX-2. (b) Precise GLIFT logic for MUX-2. (c) Simplified GLIFT logic for MUX-2.

Figure 6 (a) shows the synthesized netlist for MUX-2. It consists of two AND-2, an OR-2 and an inverter. Figure 6 (b) shows the precise GLIFT logic of MUX-2 after mapping the design netlist to a GLIFT library. We have observed that the tracking logic for OR-2 in Fig. 6 (b) can be replaced by the tracking logic for XOR-2. Let A, B, O be the inputs and output of OR-2 and XOR-2. Use A_t, B_t and O_t to denote their security labels. The GLIFT logic for OR-2 and XOR-2 are shown in Equations (4) and (1) respectively.

By comparison of Equations (4) and (1), mapping the OR-2 to the GLIFT logic for XOR-2 in the MUX-2 example may lead to further simplification of the entire GLIFT circuit. In addition, logic equivalence checking shows that the simplified GLIFT logic for MUX-2 is functionally equivalent to the precise one. Thus, the GLIFT logic for MUX-2 has been simplified without adding false positives.

Similarly, one can selectively map the logic primitives in the design netlist to alternative simplified versions of tracking logic in the GLIFT library. However, it is important to understand the tracking logic for which gates can be replaced without introducing a significant amount of false positives.

In order to map an OR-2 gate to the tracking logic for XOR-2 in the GLIFT library, we need to search for OR-2 gates that can be replaced by XOR-2 without changing the functionality of the original hardware design. Although the truth tables (`0111` for OR-2 and `0110` for XOR-2) for these two gates differ from each other, such a replacement is still possible due to internal don't-cares. Consider the MUX-2 example, the input combination `11` will never be observed at the OR-2 gate due to correlation caused by reconvergent fanout. In this case, the truth table for OR-2 can be reduced to that for XOR-2. Similarly, one may reduce the truth table for AND-2 (`0001`) to that for NXOR-2 (`1001`), whose GLIFT logic can also be described as Equation (1).

Now that we understand logic primitives can possibly be mapped to a simpler version of tracking logic in the GLIFT library, an additional step would be searching for logic primitives with internal don't-cares, which is a well-studied problem in logic synthesis. It is necessary to point out that there can be a few gate pairs both with don't-cares but cannot be replaced at the same time (the don't-care condition for one gate is dependent on the other). If we replace both gates in such pairs, it may cause a small amount of false positives. We will show how this method can lead to significant simplification of GLIFT logic while adding a minimum amount of false positives in the experimental results section.

4.4 Quantitatively Introducing False Positives

Some information flow security properties can take a long time to prove on a precise GLIFT model, e.g., proving potential key leakage caused by a Trojan triggered under a rare condition. A precise GLIFT model will indicate no harmful flow of information in most cases, resulting in a false conclusive proof. In such cases, the proof process can be significantly accelerated by creating imprecise GLIFT models with a certain amount of false positives. Figure 7 illustrates such a solution.

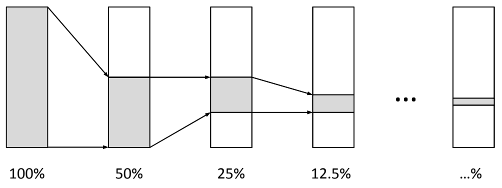

Figure 7: Creating imprecise GLIFT models with different amounts of false positives. The gray parts show the gates mapped to imprecise GLIFT logic.

We start from the least precise GLIFT model by mapping all gates in the design netlist to the most imprecise version of tracking logic. This will cause a significantly higher amount of information to flow to the output and allow formal tools

Table 1: Complexity and precision of GLIFT logic created using different synthesis commands in *ABC*.

Benchmarks	Area							False positives						
	orig	bal.	bdd	resyn	resyn2	comp.	comp.2	orig	bal.	bdd	resyn	resyn2	comp.	comp.2
alu2	4076	3754	3505	3384	3195	3384	3192	1.65%	1.65%	–	1.66%	1.74%	1.65%	1.64%
alu4	7226	6818	20093	6291	5799	6274	5602	2.00%	2.00%	–	2.02%	2.13%	2.02%	2.14%
apex6	5200	5074	21987	4967	4926	4936	4905	0.78%	0.78%	–	0.83%	0.83%	1.04%	1.04%
C1908	2106	2057	521783	2020	1826	1928	1972	19.70%	19.70%	–	19.70%	19.70%	19.70%	19.70%
C2670	3675	3938	25967	3701	3870	3748	3808	2.95%	1.68%	–	1.45%	1.47%	1.45%	1.47%
C432	1494	1488	22540	1338	1290	1417	1243	6.31%	6.31%	–	6.29%	6.29%	6.28%	6.25%
C880	2292	2220	177382	2226	2196	2145	2164	1.28%	1.28%	–	1.27%	1.27%	1.28%	1.27%
frg2	12268	11838	61429	7400	5128	7269	5018	2.16%	2.16%	–	2.19%	2.19%	2.30%	2.26%
k2	13612	11358	446236	10033	9899	9873	9674	0.43%	0.43%	–	0.42%	0.42%	0.42%	0.42%
rot	7079	6901	82054	5231	4351	5087	3927	2.27%	2.47%	–	2.21%	1.08%	2.19%	1.67%
x3	8241	8370	36963	5446	4506	5541	4813	1.06%	1.06%	–	1.09%	2.38%	1.09%	2.37%
x4	4736	4619	11205	2885	2709	2878	2781	0.80%	0.80%	–	0.88%	0.90%	0.88%	0.81%

to capture a possible violation in a shorter amount of time. However, we need to further check this is a *bona fide* security issue rather than a false alarm. This can be done by gradually decreasing the amount of imprecise gates and seeing if the violation still arises. If formal tools continue to report the same violation when only a small amount of gates in the GLIFT model are imprecise, we have some confidence that it is a real security issue. In the meanwhile, we can localize the security vulnerability that causes the violation to a relatively smaller portion of the entire design.

5. EXPERIMENTAL RESULTS

This section presents our experimental results. Section 5.1 shows how different logic synthesis optimizations lead to variations in precision and complexity of the GLIFT logic. Then, Section 5.2 shows how GLIFT logic can be simplified while introducing no, or a small amount of false positives, and Section 5.3 demonstrates that these false positives can result in reductions of security verification time.

5.1 Logic Optimization Analysis

To show the effect of logic optimizations on the precision and complexity of GLIFT logic, we use different *ABC* [16] commands to synthesize *IWLS* benchmarks [11] for GLIFT logic generation. We use area as the measure of complexity and quantify precision using the percentage of false positives.

We tested the *balance*, *bdd*, *resyn*, *resyn2*, *compress*, and *compress2* commands. The *balance* command is frequently used to create a balanced logic network before further optimization; *bdd* can be used to generate false positive free GLIFT logic [10]. *resyn2* (*compress2*) performs similar optimization to *resyn* (*compress*) but using higher effort. The synthesized logic networks are directly mapped to the GLIFT library in order to eliminate the effect of technology mapping on GLIFT logic generation. The GLIFT logic is synthesized using the *resyn2* command and mapped to the *mcnc* library. Then we perform simulation across 2^{20} random vectors generated by a linear feedback shift register (LFSR) using *ModelSim* to count the percentage of false positives.

Table 1 shows how different synthesis commands can lead to variations in complexity and precision of the GLIFT logic. The GLIFT logic generated using the *bdd* command is always the most precise and thus used as the baseline for counting false positives. However, such precision comes at significant cost in complexity. The GLIFT logic generated using the *balance* command is close to that generated from the original design in both precision and complexity. By comparison of GLIFT logic generated using *resyn* and *resyn2* (also *compress* and *compress2*), a higher optimization effort tends to reduce the complexity of GLIFT logic while introducing additional false positives.

5.2 Optimized Mapping Analysis

We use the methods from Sections 4.2 and 4.3 to simplify the GLIFT logic. A GLIFT library containing different versions of tracking logic for AND-2 and OR-2 created using the method introduced in Section 4.1 is maintained for our test. We use *Yices* to solve QBFs and *ABC* to search for don't-care conditions in the design netlist.

We perform precision and complexity analysis in a similar way to Section 5.1 and count the total number of times for the output labels to be logical `1`. We use this count value as a measure of tainted flows. Table 2 shows complexity and precision of GLIFT logic before and after simplification using several *IWLS* benchmarks [11].

Take the *des* benchmark as an example, the area of the original GLIFT logic is 32295. By comparison, the areas of the simplified GLIFT logic are 26665 and 25476 respectively. The precise and imprecise methods updated 2120 and 1785 out of the 4072 gates in the design netlist respectively. There are 17.4% and 21.1% reductions in area for the two solutions. The imprecise solution adds only 0.05% of false positives. On average, there is 14.47% reduction in area while adding only 0.20% of false positives. We can see that the GLIFT logic can be simplified with only a small amount of false positives added. The reduction in complexity of GLIFT logic will lead to improved performance in information flow security verification.

5.3 Verification Time Analysis

We use four RSA cores for verification time analysis. We choose RSA because it is possible to verify the same security property on similar designs of different sizes for comparison. We first use a precise GLIFT library to create tracking logic for these cores and verify the security property that the secret key leaks to the ciphertext ready output. For all the cores tested, the property cannot be proved within 10 minutes. We then use the method described in Section 4.4 to make a certain percentage of gates imprecise and create several imprecise GLIFT circuits. We prove the same security property on these imprecise GLIFT circuits to see how our technique can speed up information flow security verification. Figure 8 shows the verification time results.

From Fig. 8, larger RSA designs with the same percentage

Table 2: Complexity and precision of GLIFT logic before and after simplification.

Benchmarks	Area			Gates/Updated			Tainted flows		False positives
	orig	precise	imprecise	orig	precise	imprecise	precise	imprecise	
alu2	2480	2211	2174	337	113	90	4903084	4908794	0.12%
alu4	4706	4081	3920	701	291	230	6769210	6781287	0.18%
C3540	17014	14876	14239	2477	851	733	18950789	18959193	0.04%
C5315	24019	21160	20318	3706	1422	1151	88917526	89558249	0.72%
C7552	20783	18260	17696	3693	1716	1393	78408952	78868299	0.59%
des	32295	26665	25476	4072	2120	1785	193447264	193535901	0.05%
i10	19912	18082	17392	3119	951	810	121081605	121179641	0.08%
pair	13860	13613	13111	2175	446	420	90218906	90766169	0.61%
t481	231	129	120	47	17	14	983091	983091	0.00%
too_large	2118	2067	2094	281	7	7	1493984	1493984	0.00%
ttt2	1108	1073	1031	168	29	29	13436887	13436887	0.00%
x1	2132	2071	2073	300	4	4	17616423	17616423	0.00%

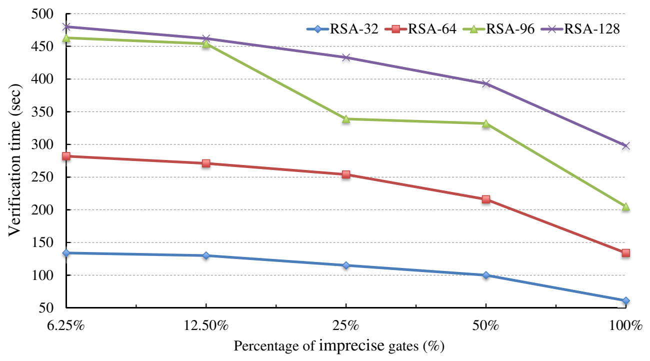

Figure 8: Verification time for different sized RSA cores with different percentages of imprecise gates.

of imprecise gates tend to take longer to verify. For the same RSA design, making a higher percentage of gates imprecise can lead to more significant speedup of the verification process. In design practice, some security properties may only be conclusively proved after a certain number of time frames. However, such incomplete proofs may mean some security vulnerabilities lurk in the design. It may be beneficial to start the proof from an imprecise model to identify a potential security issue. One can then gradually increase the model's precision and determine if it's a false alarm. Although our approach requires running multiple verifications, which may introduce additional design overheads, it still can be beneficial if a security property takes a long time to prove on the precise GLIFT logic. Our method allows a quick profiling of potential security vulnerabilities. There can be significant reduction in verification and debugging time if we can use the knowledge from the initial profiling to restrict our analysis to a smaller region of the design.

6. RELATED WORK

There are many works that address security problems using information flows. Denning was amongst the first to take an information theory approach to reason about security [21]. McLean [17] and Gray [8] pioneered the formalization of security properties using an information flow model. More recent research work focuses on employing information flow analysis across different layers of the computer system stack [22, 13, 23, 25, 28, 29]. A number of these use information flow analysis to build secure hardware.

It has been argued that using the correct abstraction is an important factor in reducing the complexity of the security analysis [15]. For instance, we could model the system at the register transfer level (RTL), and use RTL information flow analysis tools such as [14, 28, 29]. This would allow a designer to assign a one bit label to an entire variable – that would make verification faster, but the results would not present any bit level flows. However, there are many scenarios that require a lower level (gate level) IFT model, e.g., detecting some hardware Trojans [9]. Picking the correct abstraction is an important decision for system security modeling, and one that provides a complementary approach that could be used to further tradeoff between the precision and complexity ideas that we present in this work.

There is some relevant work in the logic optimization domain. Mishchenko et al. use the complete don't-care set for logic optimization [18]. Their technique uses a SAT solver to compute a complete don't-care set in local reconvergent fanout regions and leverages these don't-care conditions to optimize the design. As discussed in Section 3, logic synthesis tools will significantly change the structure of the design. As a result, the security labels of internal signals may be synthesized away. Our technique preserves the security labels while optimizing the design.

7. CONCLUSION

This work discusses the tradeoffs in precision and complexity of hardware IFT during logic synthesis and library mapping. It reveals how logic synthesis can affect the precision in two distinct directions. In addition, it proposes two novel methods that leverage the internal don't-care conditions to reduce the complexity of GLIFT logic while adding a minimum amount of false positives. It also provides a technique to allow quantitatively introduction of false positives in order to accelerate the security verification process.

8. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant NSF CNS-1527631.

9. REFERENCES

[1] S. Adee. The hunt for the kill switch. *Spectrum, IEEE*, 45(5):34–39, May 2008.

[2] N. Balabanian and B. Carlson. *Digital Logic Design Principles*. Wiley India Pvt. Limited, 2007.

[3] A. Becker, D. Maksimovic, D. Novo, M. Ewaida, A. G. Veneris, B. Jobstmann, and P. Ienne. Fudgefactor: Syntax-guided synthesis for accurate RTL error localization and correction. In *11th International Haifa Verification Conference, HVC'15*, pages 259–275, Nov. 2015.

[4] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson. Stealthy dopant-level hardware trojans. In *the 15th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'13, pages 197–214, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] D. E. R. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11, April 1982.

[8] J. W. Gray III. Toward a mathematical foundation for information flow security. *Journal of Computer Security*, 1(3):255–294, 1992.

[9] W. Hu, B. Mao, J. Oberg, and R. Kastner. Detecting hardware trojans with gate-level information-flow tracking. *Computer*, 49(8):32–40, August 2016.

[10] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 7(3):1067–1080, June 2012.

[11] IWLS. Iwls benchmarks ver. 3.0, 2005. http://iwls.org/iwls2005/benchmarks.html.

[12] R. Kastner, J. Oberg, W. Hu, and A. Irturk. Enforcing information flow guarantees in reconfigurable systems with mix-trusted IP. In *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[13] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007.

[14] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Position paper: Sapper – a language for provable hardware policy enforcement. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 39–44, New York, NY, USA, 2013.

[15] X. Li, M. Tiwari, B. Hardekopf, T. Sherwood, and F. T. Chong. Secure information flow analysis for hardware design: Using the right abstraction for the job. In *the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 8:1–8:7, New York, NY, USA, 2010.

[16] Berkeley Logic Synthesis and Verification Group. Abc: A system for sequential synthesis and verification, release 10216. http://www.eecs.berkeley.edu/~alanmi/abc.

[17] J. McLean. Security models and information flow. In *1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 180–187, 1990.

[18] A. Mishchenko and R. K. Brayton. Sat-based complete don't-care computation for network optimization. In *Design, Automation and Test in Europe*, pages 412–417, March 2005.

[19] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. *the 10th International Conference on Principles and Practice of Constraint Programming, CP'04*, chapter Understanding Random SAT: Beyond the Clauses-to-Variables Ratio, pages 438–452. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[20] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Information flow isolation in I2C and USB. In *Design Automation Conference (DAC)*, pages 254 –259, June 2011.

[21] D. E. Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan 2003.

[23] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGOPS Oper. Syst. Rev.*, 38(5):85–96, Oct. 2004.

[24] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 189–200, New York, NY, USA, 2011.

[25] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *international conference on Architectural support for programming languages and operating systems*, ASPLOS'09, pages 109–120, New York, NY, USA, 2009.

[26] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[27] R. Waugh. Could a vulnerable computer chip allow hackers to down a boeing 787? 'back door' could allow cyber-criminals a way in, May 2012. http://www.dailymail.co.uk/sciencetech/article-2152284/.

[28] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012.

[29] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015.