Property Specific Information Flow Analysis for Hardware Security Verification

Wei Hu
Northwestern Polytechnical Univ.
Xi'an 710072, China
weihu@nwpu.edu.cn

Armaiti Ardeshiricham
University of California, San Diego
La Jolla, CA 92093, USA
aardeshi@eng.ucsd.edu

Mustafa S Gobulukoglu
University of California, San Diego
La Jolla, CA 92093, USA
mgobuluk@ucsd.edu

Xinmu Wang
Northwestern Polytechnical Univ.
Xi'an 710072, China
wangxinmu@nwpu.edu.cn

Ryan Kastner
University of California, San Diego
La Jolla, CA 92093, USA
kastner@ucsd.edu

ABSTRACT

Hardware information flow analysis detects security vulnerabilities resulting from unintended design flaws, timing channels, and hardware Trojans. These information flow models are typically generated in a general way, which includes a significant amount of redundancy that is irrelevant to the specified security properties. In this work, we propose a property specific approach for information flow security. We create information flow models tailored to the properties to be verified by performing a property specific search to identify security critical paths. This helps find suspicious signals that require closer inspection and quickly eliminates portions of the design that are free of security violations. Our property specific trimming technique reduces the complexity of the security model; this accelerates security verification and restricts potential security violations to a smaller region which helps quickly pinpoint hardware security vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; **Information flow control**; *Formal security models*;

KEYWORDS

Hardware security, security verification, information flow analysis, security property, design methodology

1 INTRODUCTION

Hardware security vulnerabilities stemming from performance design optimizations [12, 15], timing channels [1], and hardware Trojans [22] create attractive new attack surfaces to hackers and can render strong software security protection techniques useless. There are a large number of mitigations for these vulnerabilities including ARM TrustZone, Apple Secure Enclave, and Intel SGX. However, the security of these techniques and their implementations are notoriously hard to verify. Even the smallest change or incorrect implementation of the mitigation technique can lead to a severe security flaws [17].

Hardware information flow tracking (IFT) [3, 4, 24, 27] provides a general technique for secure hardware design and verification, which can help identify and eliminate potential security vulnerabilities related to timing channels [2], hardware Trojans [9, 11], and access control [19]. However, most existing information flow analysis methods tend to take a general approach where the information flow models (referred to as *IFT models* for simplicity hereafter) are created to track every flow through the hardware. This is powerful as it allows us to prove any IFT-related security property using this model. But we typically only look at a small number of flows related to the properties that we aim to verify. We leverage this idea and take the security properties into account when generating customized IFT models – an approach that we call *property specific information flow analysis*.

We take advantage of the fact that a specific security property typically only needs to consider a small portion of the design. A general IFT model good for verifying different properties contains a significant amount of redundancy when considering a specific set of properties. As an example, a common property aims to detect if a cryptographic key leaks to an unintended location. As a first pass, we could see if a smaller subset of the key bits leaks; this would be done in a much faster way than checking if any of the bits of the key leak information since the resulting IFT model will be drastically simplified. We show that leveraging the security properties improves verification performance.

In this work, we propose a property specific solution to information flow security. We take into account the security properties during IFT model generation and thus can create IFT models tailored (and optimized) for different properties. Our method has benefits in simplifying the IFT model, improving security verification performance, and pinpointing security vulnerabilities. Specifically, we make the following contributions.

- Proposing a property specific approach to enforcing information flow security;
- Providing techniques for performing property specific IFT model optimization;
- Presenting experimental results to demonstrate the verification performance benefits of the proposed method.

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240839

The reminder of this paper is organized as follows. We briefly review related work in Section 2 and provide our threat model in Section 3. Section 4 covers information flow and precision tradeoff security properties. In Section 5, we describe our property specific information flow analysis method in detail. We present experimental results in Section 6 and conclude the paper in Section 7.

2 RELATED WORK

We categorize the related work into hardware security vulnerability detection, information flow analysis for secure hardware design, and property driven security and safety verification.

We divide hardware security vulnerability detection methods into three categories: functional verification, side channel analysis, and security verification. Functional verification methods primarily focus on activating paths of low controllability [21], detecting suspicious signals [26], or identifying (nearly) unused circuit [7, 28] to uncover malicious functionality. Side channel analysis employs statistical and information theoretical metrics to analyze side channel measurements such as path delay and power consumption to extract fingerprints of hardware Trojans [18] or to quantify the amount of leakage [16]. Security verification methods formally prove adherence to security properties and indicate a security issue when the property is violated [5, 9, 11].

Information flow analysis is a frequently used technique for hardware security verification. Tiwari et al proposed the gate level information flow tracking (GLIFT) technique to precisely measure all logical information flows at the level of Boolean gates [24]. Successive research work has demonstrated the effectiveness of GLIFT for crafting secure architectures [23], identifying timing channels [19, 20] and detecting hardware Trojans [9]. To overcome the high design overheads of the original GLIFT method, various approaches have been proposed to enforce information flow security at higher levels of abstraction [3, 11]. Other researchers incorporate type systems into hardware description languages for verifying hardware security from the perspective of information flow [4, 13, 14, 27]. However, these methods tend to create a unified IFT model suitable for verifying multiple security properties, which can incur significant irrelevant redundancy when only specific properties are being verified.

There is a recent move towards property driven security verification inspired by the success of assertion languages (e.g., Verilog SVA and VHDL PSL) in functional verification. Hu et al proposed a similar property driven approach for verifying hardware security [8]. In addition to the traditional functional specification and synthesis flow, security properties are specified, refined, and verified as the design is translated to lower levels of abstraction. Urdahl et al [25] proposed a top-down design methodology, where abstract properties are automatically generated from a system-level description and later refined along with the design process for formal verification of safety properties such as susceptibility of faults. Existing work focuses on how to refine security and safety properties across different levels of abstraction, while this paper investigates how to create property specific IFT models in order to accelerate hardware security verification.

3 THREAT MODEL

We target hardware security vulnerabilities that can be modeled using information flow, e.g., leaking sensitive information (e.g., the secret key), memory and resource isolation, cycle-based timing channels, and hardware Trojans. We focus on logical and timing information flows and do not consider flows of information flows through power and electromagnetic side channels. We assume access to the hardware design, e.g., register transfer or gate level netlist, which is necessary to create an IFT model to perform security verification. We consider the case when the security verification is performed during the hardware design process and is aimed at identifying and eliminating security vulnerabilities before the hardware is fabricated.

4 SECURITY PROPERTIES

4.1 Information Flow Security Properties

Information flow security properties are derived from the notion of *non-interference* [6], which states that `high` (e.g., confidential or untrusted) information should never flow to a `low` (e.g., unclassified or trusted) security domain. Frequently used information flow security properties include *confidentiality*, *integrity*, and *isolation*.

Confidentiality properties insure that sensitive (`high`) data never flow to a (`low`) variable. As an example, the secret key should not flow to the ciphertext ready signal otherwise there would be a timing side channel which is stated as:

```
1: set key := high
2: assert ready == low
```

Integrity is the dual of confidentiality. Here, we mark untrusted hardware resources with a `high` label and verify that they do not affect critical memory locations (labeled as `low`). For example, the program counter (pc) should not be overwritten by data from unprotected network packet:

```
1: set ethernet_data := high
2: assert pc == low
```

Isolation can also be enforced as an information flow security property. This states that there should never be information exchange between two hardware components of differing trust. For example, trusted (`low`) IP cores sitting in the secure world should be separated from those untrusted (`high`) in the insecure world in SoC designs:

```
1: set crypto_core := high
2: assert lcd_ctrl == low
```

The above security properties are effective in preventing unintended flows of information. However, in realistic systems, `high` information may be allowed to flow to a `low` portion under certain circumstances, e.g., during debug mode or when they have proper security protection (perhaps using an encryption protocol). Figure 1 shows some examples of mode specific information flows.

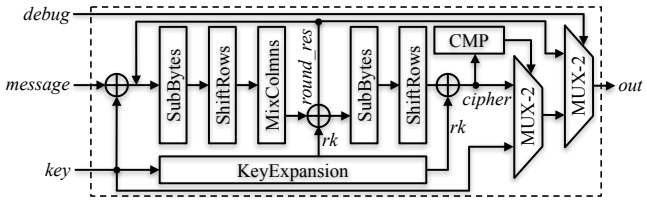

Figure 1: Examples for flow mode security properties

In this AES implementation, the intermediate encryption results, i.e., *round_res*, are allowed to flow to *out* when the core is working in debug mode, but this is prohibited under normal operation. The ***when*** keyword provides a way to specific a conditional security property that only allows information to flow under specific working modes.

```
1: set round_res := high
2: assert out == low when !debug
```

Consider another scenario where the key is safe to flow to the ciphertext (since this is mathematically known as a secure one-way function) while it should not flow to another location. Since there is always a flow from the key to the ciphertext in cryptographic functions, we need a way for declassification (i.e., saying that flows through the cipher are okay) The ***allow*** keyword provides a way to manually declassify a signal and check if there is another flow.

```
1: set key := high
2: assert out == low allow cipher
```

The mode specific information flow security properties allow some relaxation on safe information flows. The *when* property seamlessly translates to the *disable if* statement of assertion languages. The *allow* property can be mapped by declassifying the security label of the specified path. In the AES example, we can declassify the label of *cipher* assuming that it is properly protected by the AES algorithm, which will expose the harmful flows of information through the hardware Trojan.

4.2 Precision and Complexity Modeling Tradeoffs

IFT models should never allow false negatives, which means that they fail to identify a security property violation. On the other hand, it is safe (though not ideal) to have false positives, which indicates non-existent flows of information. Such false positives can be employed to reduce the complexity of IFT models and accelerate security verification.

Take the two-input AND gate (AND-2) as an example. We can create various IFT models for AND-2 with different levels of precision by adding different amounts of false positives. Use A, B and O to denote the inputs and output of AND-2; At, Bt and Ot are their security labels respectively. Let the labels be logical 1 (or 0) when the variables carry high (or low) information. Equation (1) gives the precise IFT model for AND-2. It outputs logical 1 *if and only if* high information flows to O.

$$Ot = ABt + BAt + AtBt \tag{1}$$

Such precise IFT model accurately measures all actual flows of information. However, sometimes it is desirable to relax the precision a bit by adding false positives (i.e., non-existent information flows) to allow a quick profile of potential security flaws. Consider a hardware Trojan that leaks secret information when activated. Under a precise IFT model, a high information flow can be captured only when the Trojan is triggered. By comparison, an imprecise IFT model can bypass the trigger condition and always allow high information to flow, enabling faster detection of the harmful flow.

Introducing false positives to the precise IFT model is equivalent to adding minterms to the Boolean function of the IFT model. If done in an intelligent manner, this can make the model less complex and accelerate security verification. Don't care based optimization achieves both goals. When setting A and B to don't care, the resulting simplified IFT model is shown in Equation (2). This is the most conservative yet reasonable IFT model for AND-2. It states that when either of the inputs contains high information, the output will be high. A conservative IFT model does not consider the effect of input values on label propagation and always allows high information to propagate.

$$Ot = At + Bt \tag{2}$$

Similarly, we can use this method to construct IFT models of different precision and complexity for other standard cells. In this way, we can create an IFT library with primitive IFT models for precision and complexity tradeoffs. Based upon such an IFT library, we define the ***use*** keyword to specify precise or imprecise IFT model will be instantiated. Figure 2 illustrates the idea of instantiating IFT models of different precision for security verification.

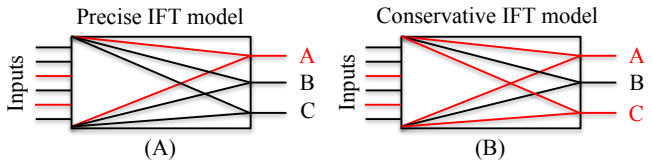

Figure 2: Instantiating IFT models of different precision for security verification. The precise IFT model exactly models the flows (no false positives or false negatives). A conservative model would state that there are additional flows (false positives and still no false negatives).

Here, both the precise and conservative IFT models detect a flow from the high inputs (in red) to output A while no high information flow to B. These two security properties are easy to verify. The subtler case is when the precise IFT model indicates no high information flow to C under affordable verification cost but the conservative IFT model quickly captures one. This can be either a false positive, e.g., a disabled debug port that had access to the secret key or a security vulnerability that only occurs under a rare condition, e.g., a hardware Trojan triggered under a rare event to leak information. However, there is no easy solution to distinguish a false positive from a security violation if the property cannot be verified at acceptable verification cost. We will provide some

methods for eliminating false positives and pinpointing security vulnerabilities in Section 5.3.

We can strike a balance between precision and complexity by instantiating conservative but simpler IFT models during the initial verification stages to allow fast detection of potential security violations. This will also help exclude design portions free of security violations and restrict our analysis to design regions that may contain a security flaw. Afterwards, we can instantiate more precise but more complex IFT models for that region to check if the detected security violations were false positives or actual security property violations.

5 PROPERTY SPECIFIC INFORMATION FLOW ANALYSIS

5.1 Design Methodology

Figure 3 shows the design flow for property specific information flow analysis. Our method creates IFT models tailored and optimized towards the given properties. The security properties are compiled to security constraints and assertions. These are combined with the trimmed IFT model to run security analysis using formal verification, simulation, and/or emulation.

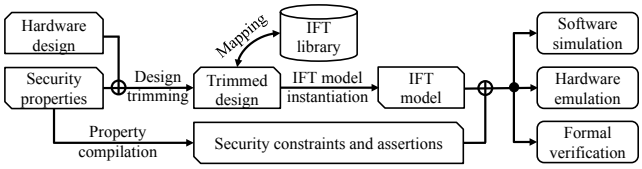

Figure 3: Design flow of property specific information flow analysis.

It is important to understand the difference between existing information flow analysis methods and the proposed approach. Existing methods first create an IFT model with $2n$ inputs (i.e., n original inputs and their security labels) [3, 24]. It then reduces the IFT model under a given property for security verification. Such a reduction problem has $O(2^{2n})$ complexity. The property specific approach first solves an $O(2^n)$ reduction problem on the original design and then creates an IFT model tailored for the given security property through linear time IFT library mapping.

5.2 Property Specific IFT Model Optimization

We create IFT models tailored to given security properties through *fanout* and *fanin* trimmings as shown in Fig. 4.

We start with fanout trimming. In this step, we perform a graph search from the source of `high` information until reaching the primary outputs. All of the nodes (***logic primitives and signals***) visited in the forward search are added to the fanout set (between the blue dashed lines) of the `high` input. The nodes that are never visited during fanout analysis will be labeled as `low` since they will never be influenced by the `high` input. We use a *depth first search* strategy in fanout search and terminate when we reach the checkpoint so that only a minimum number of property specific paths will be added to the resulting IFT model.

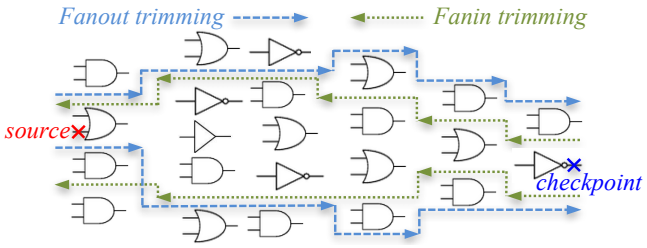

Figure 4: Property specific trimmings for simplifying the IFT model.

The fanin trimming stage performs a backward graph search from the security checkpoint specified in the security property until it reaches the primary inputs. All the nodes visited in backward search are added to the fanin set (between the green dashed lines) of the checkpoint. The nodes that are never reached can be trimmed away for the given property. When a suspicious signal identified during fanout trimming is trimmed away in fanin search, we can specify a new property for that signal and re-run both trimmings in case we need to keep the suspicious signal for further security verification.

After performing both trimmings, we can partition the circuit nodes for IFT model generation. Nodes out of the fanin set will be removed directly. If a node is in the fanin set but outside the fanout set, only the original design functionality needs to be imported. If a node is in the intersection of the fanin and fanout sets, both the original functionality and additional IFT model for label propagation are necessary. The property specific trimming steps eliminates nodes irrelevant to the given property from the IFT model.

Intuitively, the fanout trimming allows a quick and safe profiling of where `high` information can possibly flow. It helps identify suspicious signals for further analysis, e.g., using these signals to specify more pertinent security properties and also exclude design portions where `high` information never propagates, which restricts our analysis to a smaller region of the design. If the security checkpoint is out of the fanout set, the security property automatically holds (as no flow is possible). The fanin trimming enables an understanding of which source nodes can affect the security checkpoint. The nodes out of the fanin set are trimmed to simplify the IFT model.

5.3 Property Specific Precision and Complexity Tradeoffs

A first way to tradeoff between precision and complexity is at the level of logic primitives (i.e., Boolean gates). As shown in Section 4.2, we can create multiple IFT models of different precision and complexity for each logic primitive and then use our precision security property to specify which IFT model will be instantiated for a given logic primitive.

Additionally, we can perform precision and complexity tradeoffs at a coarser granularity of modules. The precision security property can globally specify the precision level of IFT models for all the logic primitives within a module while allowing different precision levels across different modules. Such tradeoff is desirable for functions with good amount of diffusion such as cryptographic cores where each `high` input essentially flows to all outputs. Instantiating a

less precise IFT model for the function will significantly reduce its security verification complexity.

With various precision and complexity tradeoff methods, a binary search strategy can be adopted for pinpointing security vulnerabilities. Consider a hardware design with a Trojan that is triggered under a rare condition to leak information. There will be no high information flow until the Trojan is activated under the precise IFT model. By comparison, the leakage can be captured in no time when the imprecise IFT model is instantiated. The binary search method creates two IFT models for the design with 50% (of modules or logic primitives) instantiating the precise model and 50% the imprecise. One will indicate flows of high information while the other will not. This restricts the security vulnerability (i.e., the Trojan) to the 50% portion of the design, which when made imprecise will indicate a security violation. We then continue our search from that portion by further dividing it to halves. This progressively restricts the security vulnerability to smaller regions of the design.

6 EXPERIMENTAL RESULTS

In this section, we present experimental results. We first show how our property specific information flow analysis method can improve security verification performance through formal proof, simulation, and emulation. We then demonstrate how precision tradeoff properties can accelerate security verification and help pinpoint security vulnerabilities.

6.1 Formal Verification Analysis

We use several benchmarks from *Trust-HUB.org* and prove different security properties using *Mentor Graphics Questa Formal* for formal verification analysis. We generate IFT models trimmed for specific properties for formal proof and also create the (untrimmed) original IFT model for comparison. The original IFT model will import the constraints derived from the same properties for simplification. Table 1 shows the benchmarks, security properties and proof results. Here, the run-time results include the time for formal model compilation and proof; IFT model trimming adds negligible performance overheads over the original IFT model generation. In our proof, we use four cores to run verification in parallel and set the timeout limit to 10 minutes.

For the AES-T100 benchmark, we verify the security property that the lowest key bit (i.e., *key[0]*) should flow to the ciphertext when an encryption is completed. We mark *key[0]* as high and assert the lowest ciphertext bit (i.e., *cipher[0]*) should be high after 22 clock cycles, which is the time needed to encrypt a message. The property is proven to hold under both the original and trimmed IFT models with proof times of 461 and 341 seconds respectively.

The AES-T400 benchmark contains a hardware Trojan triggered when the 128 plaintext bits are all logical 1. When activated, the Trojan leaks the key bits to an unused pin named *Antena* (sic). We label the lowest key bit (i.e., *key[0]*) as high and use our fanout trimming technique to profile where this key bit could possibly flow. Trimming analysis shows that *key[0]* could flow to both the *ciphertext* and *Antena* outputs. Then, we focus the verification process on *Antena*. We assert *Antena* as always low and attempt to formally prove this property using precise IFT models. The property could not be verified using either the original or trimmed IFT model within the proof time limit (set at 10 min). We then replace the precise IFT models with conservative ones as discussed in Section 4.2 and re-run the proof. The property fails to hold after 208 and 5 *sec* respectively for the original and trimmed IFT models, indicating that *key[0]* could flow to *Antena*. We also take an internal signal, namely *TSC_SHIFTReg[0]* identified in fanout trimming and check if *key[0]* could flow there under precise IFT models. Proof results show that the property is violated after 278 and 5 *sec* respectively for the original and trimmed IFT models. The formal tool also returned a counter example showing that when all the plaintext bits are logical 1, *key[0]* will flow to the *TSC_SHIFTReg[0]* register. We can use *TSC_SHIFTReg[0]*, which is much closer to *Antena*, instead of *key[0]* as a starting point to check if the security violation under the conservative IFT model is a false positive.

The AES-T1100 benchmark contains a Trojan that XOR modulates the secret key using a random sequence. The modulated sequence is then leaked through the signal *Capacitance*. We first label the lowest key bit (i.e., *key[0]*) as high and use our fanout trimming technique to profile the fanout region of *key[0]*. Analysis results show that it flows to *Capacitance[7:0]*. We assert *Capacitance[0]* is low under both the original and trimmed precise IFT models. The proof fails after 273 and 4 *sec* respectively. It is very likely that *key[0]* will leak to *Capacitance[0]* since we instantiated precise IFT models in the proof. We further mark *key[1]* as high and assert *Capacitance[0]* is low using precise IFT models. The property is proven after 282 and 0 *sec* for the original and trimmed IFT models respectively. The trimmed IFT model has a zero proof time because *Capacitance[0]* is reduced during fanout trimming. From the AES-T1100 example, our method will automatically exclude the safe design regions under given properties so that we can focus on the remaining portions that need further security analysis.

For RSA-T100, we first mark the lowest key bit *inExp[0]* as high and check if the ciphertext ready (i.e., *ready*) signal is always low under a conservative IFT model. The property is proven within 162 and 0 *sec* for the original and trimmed IFT models respectively. The trimmed IFT model has a proof time of zero since the *ready* signal is eliminated by our trimming technique and thus no proof needs to be run (i.e., there is no connection between *inExp[0]* and *ready*). This well agrees with the RSA implementation in that the lowest key bit assumed to constant 1 (the RSA cipher requires the key to be an odd number) in the design and thus never has an influence on the encryption time. We then label the second bit of the key (*inExp[1]*) as high and check a similar security property using a precise IFT model. The security property fails after 68 and 41 *sec* for the original and trimmed IFT models respectively. The verification results show that *inExp[1]* can affect the encryption time and reveal a timing channel in the RSA implementation.

RSA-T200 has a hardware Trojan that overwrites the key register with plaintext after a certain number of encyptions. We mark the second bit of the plaintext (i.e., *indata[1]*) as high and perform fanout trimming. Results show that this plaintext bit could flow to both the ciphertext and second bit of the key register, namely *count[1]*. While a flow from the plaintext to the ciphertext is desirable for RSA, a flow from the plaintext to the key register when a separate key input port is present can be suspicious. Thus, we specify a property to assert if the flow to the key register could actually happen using a precise IFT model. Formal proof results reinforced

Table 1: Formal proof results for the original and trimmed IFT models.

Benchmarks	Security Properties	Proof status	Proof time	
			Original	Trimmed
AES-T100	**set** key[0] := high; **use** IFT_model := precise; **assert** ##22 cipher[0] == high	Proven	461	341
AES-T400	**set** key[0] := high; **use** IFT_model := precise; **assert** Antena == low	Incomplete	*timeout*	*timeout*
AES-T400	**set** key[0] := high; **use** IFT_model := conservative; **assert** Antena == low	Failed	208	5
AES-T400	**set** key[0] := high; **use** IFT_model := precise; **assert** TSC_SHIFTReg[0] == low	Failed	278	5
AES-T1100	**set** key[0] := high; **use** IFT_model := precise; **assert** Capacitance[0] == low	Failed	273	4
AES-T1100	**set** key[1] := high; **use** IFT_model := precise; **assert** Capacitance[1] == low	Proven	282	0
RSA-T100	**set** inExp[0] := high; **use** IFT_model := conservative; **assert** ready == low	Proven	162	0
RSA-T100	**set** inExp[1] := high; **use** IFT_model := precise; **assert** ready == low	Failed	68	41
RSA-T200	**set** indata[1] := high; **use** IFT_model := precise; **assert** count[1] == low	Failed	23	21
RSA-T400	**set** inExp[1] := high; **use** IFT_model := precise; **assert** cipher[1] == high **when** *ready* == 1	Failed	825	194

that the key register bit can be overwritten by the corresponding plaintext bit.

RSA-T400 contains a hardware Trojan that replaces the key with a constant value to cause deny of service. We set the second key bit (i.e., inExp[1]) to high and assert that *cypher[1]* should be high when the ciphertext is ready. The security property fails after 825 and 194 *sec* for the original and trimmed precise IFT models respectively. This is because the only high information source is replaced by the Trojan with a low constant value.

The results show that property specific IFT analysis can have significant benefits in proof performance. This is because the IFT model is trimmed for the property to be verified and new security property is incorporated for IFT model precision and complexity tradeoff. However, formal verification can be expensive when a security flaw hides behind a hard-to-cover corner case. Some security properties cannot be proved within an acceptable amount of time and memory resource. Take AES-T400 for example, proving that the lowest key bit flows to the *Antena* pin cannot be completed with 10 min. Figure 5 shows the memory consumption vs. proof radius for both the original and trimmed IFT models. From the results, formal verification can consume a huge amount of memory when the property is hard to prove, e.g., about 50 GB as the proof radius increases to 1, 000, 000 for both models.

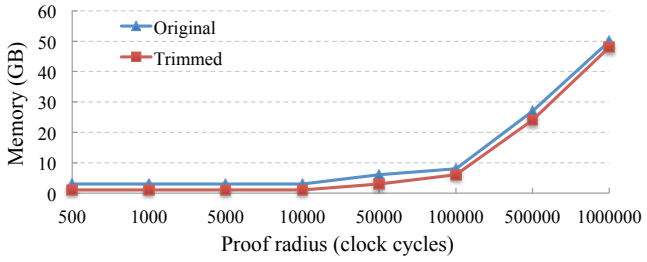

Figure 5: Memory consumption vs. proof radius for different IFT models.

In the following, we show how our property specific approach can significantly accelerate software and hardware emulation.

6.2 Simulation Analysis

We use *Mentor Graphics QuestaSim* to simulate the original and trimmed IFT models for several *Trust-HUB* benchmarks. Similarly, the trimmed IFT models are tailored for specific properties; the original IFT model are simplified using the constraints derived from the same properties. We use linear feedback shift registers (LFSRs) to generate random test vectors. Simulation terminates either when the security property is violated or after 2^{16} random tests. Table 2 shows the simulation times in *seconds*.

For the AES-T100 benchmark, the simulation times for the trimmed IFT models vary with the number of ciphertext bits analyzed while the simulation times for the original IFT model do not see large variations. When asserting the existence of flow from *key[0]* to *Antena*, the AES-T400 benchmark has simulation times of 1776 and 34 *sec* for the original and trimmed IFT models respectively. We then check if *key[0]* always flows to *cipher[0]* in AES-T400, the simulation times are 1796 and 1319 *sec* for the original and trimmed IFT models. Our trimming method also see significant simulation performance improvements in the AES-T1100 benchmark.

When the RSA-T100 benchmark is trimmed to understand the flow of *key[0]* to *cipher[0]*, the simulations indicate a security property violation after 6 and 5 *sec*, respectively. The violation is caused by the hardware Trojan in the benchmark since *key[0]* should not have an influence on *cipher[0]*. This is because the lowest key bit is assumed to be always logical 1 and hard-coded in these RSA implementations. Thus, *key[0]* does not have any effect on *ready* either in the second property. Specifically, *ready* is out of the fanout set of the *key[0]* for our trimming method, resulting a proof time of zero for the trimmed IFT model while the original IFT model has a simulation time of 2936 *sec* without a security property violation after 2^{16} encryptions. We then check a third security property for RSA-T100, requiring that *key[1]* should not flow to *ready*; the property is violated only after 7 *sec* under both IFT models, indicating a timing channel in the benchmark. For RSA-T200, the property asserting that *indata[1]* should not flow to *count[1]* has simulation times of 7 and 6 *sec* for the original and trimmed IFT models respectively. When checking if *key[1]* flows to *cipher[1]* in the RSA-T400 benchmark, the original and trimmed IFT models have identical simulation times. This is because these two IFT models have very close optimization results when compiled by *QuestaSim*.

Table 2 indicates that the trimmed IFT models significantly improve simulation performance especially for the relatively larger

Table 2: Simulation times for the original and trimmed IFT models (*sec*).

Benchmarks	Security Properties	Original	Trimmed
AES-T100	**set** key[0] := high; **use** IFT_model := precise; **assert** ##22 cipher[0] == high	1830	1238
AES-T100	**set** key[1] := high; **use** IFT_model := precise; **assert** ##22 cipher[7:0] == high	1776	1310
AES-T100	**set** key[1] := high; **use** IFT_model := precise; **assert** ##22 cipher[31:0] == high	1775	1426
AES-T100	**set** key[1] := high; **use** IFT_model := precise; **assert** ##22 cipher[63:0] == high	1860	1501
AES-T400	**set** key[0] := high; **use** IFT_model := precise; **assert** Antena == low	1776	34
AES-T400	**set** key[0] := high; **use** IFT_model := precise; **assert** ##22 cipher[0] == high	1796	1319
AES-T1100	**set** key[1] := high; **use** IFT_model := precise; **assert** Capacitance[0] == high	1417	1
RSA-T100	**set** inExp[0] := high; **use** IFT_model := precise; **assert** cipher[0] == low	6	5
RSA-T100	**set** inExp[0] := high; **use** IFT_model := precise; **assert** ready == low	2963	0
RSA-T100	**set** inExp[1] := high; **use** IFT_model := precise; **assert** ready == low	7	7
RSA-T200	**set** inExp[1] := high; **use** IFT_model := precise; **assert** count[1] == low	7	6
RSA-T400	**set** inExp[1] := high; **use** IFT_model := precise; **assert** cipher[1] == high **when** *ready* == 1	2440	2437

AES benchmarks. This is due to the positive correlation between the complexity of the IFT model and simulation time.

6.3 Hardware Emulation Analysis

While simulation consumes significantly less memory and time than a formal analysis, it cannot achieve a high coverage during a reasonable amount of time on large designs. As an alternative approach, we can use emulation platforms (e.g., large FPGAs) to evaluate the IFT models under realistic inputs.

The original IFT model is typically ~3X of the hardware design under test in terms of area overhead [10]. It is usually infeasible to implement the entire original IFT model for hardware emulation even for a moderate design. For example, the original IFT model for the AES-T400 benchmark cannot fit even on the largest Virtex-7 FPGA (resource utilization not reported due to synthesis failure). Instead, we generated a trimmed IFT model for this benchmark under the security property asserting that the *key[0]* should not flow to *Antena*. For comparison, we apply the constraints derived from the security property to reduce the original model. Specifically, we set some security labels as constants (based upon the specified property) and allow the synthesis tools to optimize the model. Table 3 shows the implementation results of both the *reduced* original and trimmed IFT models synthesized by *Xilinx ISE* to target the XC7VX485T FPFA.

Table 3: Implementation results of both the *reduced* original and trimmed IFT models for the AES-T400 benchmarks on the XC7VX485T FPGA. The synthesis time is in hh:mm:ss.

Parameters	(Reduced) Original	Trimmed
Slice registers	415	415
Slice LUTs	338	346
Slices	484	507
LUT-FF pairs	573	587
Synthesis time	02:23:47	00:02:41

From Table 3, the *reduced* original and trimmed IFT models have close implementation results under the same security property. However, it takes only two minutes for the trimmed IFT model to synthesize and implement while it requires over two hours to reduce the original, which reinforces our analysis in Section 5.1.

6.4 Property Specific Design Tradeoff Analysis

We also use the AES-T400 benchmark for property specific precision and complexity tradeoff analysis. The synthesized design netlist consists of 345 modules. Our goal is to use precision related security properties to accelerate the security verification process and help pinpoint the hardware Trojan.

Our method allows both module and logic primitive level precision and complexity tradeoffs. We focus on the module level and use security properties to globally specify the type of IFT models (i.e., precise or imprecise models as described in Section 4.2) to be instantiated for all logic primitives within a module while allowing precision variations across different modules. Table 4 shows the results. Similarly, we set the timeout limit to 10 minutes.

Table 4: Verification times for property specific precision and complexity tradeoff analysis (*sec*).

	Proof group 1		Proof group 2		Proof group 3	
Setup	1-345	None	1-174	175-345	1-88	89-345
Time	*timeout*	211	*timeout*	246	*timeout*	259
	Proof group 4		Proof group 5		Proof group 6	
Setup	1-45	46-345	1-24	25-345	1-13	14-345
Time	*timeout*	269	*timeout*	274	*timeout*	275
	Proof group 7		Proof group 8		Proof group 9	
Setup	1-8	9-345	1-5	6-345	1-4	5-345
Time	*timeout*	280	*timeout*	280	*timeout*	281
	Proof group 10		Proof group 11		Proof group 12	
Setup	1-3	4-345	1-2	3-345	1	2-345
Time	*timeout*	297	*timeout*	297	*timeout*	302

For a better understanding, each proof group in Table 4 contains two different proof setups. The numbers in the proof setups indicate the number of modules that instantiate precise IFT models. As an example, proof setup 1-174 means that the first 174 modules use precise IFT models while the remaining ones imprecise.

Consider proof group 1, in the 1-345 proof setup, *all* of the 345 modules use precise IFT models while *none* in the other proof setup.

When all the modules use precise IFT models, the proof cannot be completed within the timeout limit. When all modules use imprecise IFT models, the proof indicated a security property violation after 165 *sec*. This is significant speedup in formal verification performance. However, such violation can also be resulted from the false positives introduced by the imprecise IFT models.

To check if the security violation is a false positive, we set about half (e.g., 1 to 174) of the modules to use precise IFT models while the other half imprecise in the second proof group. The 1-174 proof setup also times out; the 175-345 proof setup can be completed in 246 *sec*. By gradually reducing the number of module that instantiate imprecise IFT models, we can restrict the potential false positive to a smaller portion of the design. If the security violation still exists even when only a few modules use imprecise IFT models, we have higher confidence that the violation is more likely a real security issue. After further analysis, we restricted the potential false positive to the first module, which is exactly where the Trojan design resides.

When we restrict the potential security vulnerability to a minimum number of modules, we can perform similar property specific tradeoffs at the level of logic primitives, allowing us to pinpoint the security vulnerability to a small number of gates.

7 CONCLUSION

We propose a property specific information flow analysis approach for hardware security verification. We identify several important information flow security properties and introduce new properties for IFT model precision and complexity tradeoffs. We take the security properties into account in order to create IFT models tailored for given properties. By performing property specific IFT model optimization and quality tradeoffs, our method provides significant verification performance benefits and can help quickly pinpoint security vulnerabilities.

ACKNOWLEDGMENTS

This work was supported in part by the Fundamental Research Funds for the Central Universities under grant 3102017OQD094, the NSF under grant 1718586 and the Semiconductor Research Corporation (SRC).

REFERENCES

[1] Alexandres Andreou, Andrey Bogdanov, and Elmar Tischhauser. 2017. Cache timing attacks on recent microarchitectures. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust*. 155–155. https://doi.org/10.1109/HST.2017.7951819

[2] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *International Conference on Computer-Aided Design*. 147–154. https://doi.org/10.1109/ICCAD.2017.8203772

[3] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition*. 1695–1700. https://doi.org/10.23919/DATE.2017.7927266

[4] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Y. Serhan Gener, Onur Demir, and Jakub Szefer. 2017. SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures. Cryptology ePrint Archive, Report 2017/193. (2017).

[5] Nicole Fern, Ismail San, and Kwang Ting Tim Cheng. 2017. Detecting hardware Trojans in unspecified functionality through solving satisfiability problems. In *Asia and South Pacific Design Automation Conference*. 598–504.

[6] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symposium on Security & Privacy*. 11–20.

[7] Matthew Hicks, Murph Finnicum, Samuel T. King, Milo M. K. Martin, and Jonathan M. Smith. 2010. Overcoming an Untrusted Computing Base: Detecting and Removing Malicious Hardware Automatically. In *IEEE Symposium on Security and Privacy*. 159–172. https://doi.org/10.1109/SP.2010.18

[8] Wei Hu, Alric Althoff, Armaiti Ardeshiricham, and Ryan Kastner. 2016. Towards Property Driven Hardware Security. In *17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 51–56.

[9] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. 2016. Detecting Hardware Trojans with Gate-Level Information-Flow Tracking. *Computer* 49, 8 (Aug 2016), 44–52. https://doi.org/10.1109/MC.2016.225

[10] Wei Hu, Jason Oberg, Dejun Mu, and Ryan Kastner. 2012. Simultaneous information flow security and circuit redundancy in Boolean gates. In *International Conference on Computer-Aided Design*. 585–590.

[11] Yier Jin and Yiorgos Makris. 2012. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *the 30th IEEE VLSI Test Symposium (VTS)*. 252–257. https://doi.org/10.1109/VTS.2012.6231062

[12] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[13] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A Language for Hardware-level Security Policy Enforcement. In *Architectural Support for Programming Languages and Operating Systems*. 97–112.

[14] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson:a hardware description language for secure information flow. In *Acm Sigplan Conference on Programming Language Design & Implementation*. 109–120.

[15] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[16] Baolei Mao, Wei Hu, Alric Althoff, Janarbek Matai, Jason Oberg, Dejun Mu, Timothy Sherwood, and Ryan Kastner. 2015. Quantifying Timing-Based Information Flow in Cryptographic Hardware. In *International Conference on Computer Aided Design*. 552–559. https://doi.org/10.1109/ICCAD.2015.7372618

[17] Bits, Please. 2016. Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption. In *http://bits-please.blogspot.com/2016/06/extracting-qualcomms-keymaster-keys.html*.

[18] Seetharam Narasimhan, Dongdong Du, Rajat Subhra Chakraborty, Somnath Paul, Francis G. Wolff, Christos A. Papachristou, Kaushik Roy, and Swarup Bhunia. 2013. Hardware Trojan Detection by Multiple-Parameter Side-Channel Analysis. *IEEE Transactions on Computers* 62, 11 (Nov 2013), 2183–2195. https://doi.org/10.1109/TC.2012.200

[19] Jason 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 Design Automation Conference*. ACM, 254–259.

[20] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. 2014. Leveraging gate-level properties to identify hardware timing channels. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 33, 9 (2014), 1288–1301.

[21] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. 2012. A Novel Technique for Improving Hardware Trojan Detection and Reducing Trojan Activation Time. *IEEE Transactions on Very Large Scale Integration Systems* 20, 1 (Jan 2012), 112–125. https://doi.org/10.1109/TVLSI.2010.2093547

[22] Sergei Skorobogatov and Christopher Woods. 2012. *Breakthrough Silicon Scanning Discovers Backdoor in Military Chip*. Springer-Heidelberg, 23–40.

[23] 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 *International Symposium on Computer Architecture (ISCA)*. 189–199. https://doi.org/10.1145/2000064.2000087

[24] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete Information Flow Tracking from the Gates Up. In *the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. 109–120.

[25] Joakim Urdahl, Shrinidhi Udupi, Tobias Ludwig, Dominik Stoffel, and Wolfgang Kunz. 2016. Properties First? A New Design Methodology for Hardware, and Its Perspectives in Safety Analysis. In *the 35th International Conference on Computer-Aided Design*. Article 84, 8 pages. https://doi.org/10.1145/2966986.2980086

[26] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: Identification of Stealthy Malicious Logic Using Boolean Functional Analysis. In *Conference on Computer; Communications Security*. 697–708.

[27] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 503–516. https://doi.org/10.1145/2694344.2694372

[28] Jie Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Qiang Xu. 2015. VeriTrust: Verification for Hardware Trust. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 34, 7 (July 2015), 1148–1161. https://doi.org/10.1109/TCAD.2015.2422836