TECHNISCHE UNIVERSITÄT WIEN

Institut für Computertechnik
Institute of Computer Technology

Master's Thesis

submitted by

Philipp Wassertheurer, BSc

Registration Number 1226073

PGSLIFT – Pattern Graph Specification Language Information Flow Tracking

In partial fulfillment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Vienna, Austria, Oktober 2021

Study code:	066 504
Field of study:	Embedded Systems
Supervisors:	Univ.Ass. Dipl.-Ing. Dr.techn. Christian Krieg Univ.Prof. Dipl.-Ing. Dr.techn. Axel Jantsch

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Copyright ©2021 Philipp Wassertheurer, BSc

If you find this work useful, please cite it using the following BibTeX entry:

```
@Thesis{Wassertheurer2021,
 type        = {Master's Thesis},
 author      = {Philipp Wassertheurer, BSc},
 title       = {PGSLIFT — Pattern Graph Specification Language Information Flow Tracking},
 school      = {Vienna University of Technology (TU Wien)},
 year        = {2021},
 address     = {Gusshausstrasse 27--29 / 384, 1040 Wien},
 month       = {October},
}
```

Contact us:

philipp.wassertheurer@alumni.tuwien.ac.at

christian@drkrieg.at

This thesis is licensed under the following license: Attribution 4.0 International (CC BY 4.0)

You are free to:

1. Share — Copy and redistribute the material in any medium or format

2. Adapt — Remix, transform, and build upon the material for any purpose, even commercially.

This license is acceptable for Free Cultural Works.

The licensor cannot revoke these freedoms as long as you follow the license terms.

The entire license text is available at: https://creativecommons.org/licenses/by/4.0/legalcode

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

Diese Arbeit stellt den Gipfel meiner akademischen Karriere dar. Sie wäre nicht ohne meine Familie entstanden die mich nun jetzt seit fast 30 Jahren in jeder nur denklicher Weise unterstützt. Ich freue mich, das Kapitel Studium zu beenden und mich in neue Abenteuer und Herausforderungen zu stürzen.

Besonderen Dank möchte ich folgenden Personen widmen. Ohne euch wäre ich nicht annähernd so weit gekommen.

- Meinen Eltern Sylvia und Gerd, die mich viel zu lange durchfütterten und mir immer das Gefühl gaben alles richtig zu machen auch wenn ich selbst oft anderer Meinung war.
- Meiner künftigen Ehefrau Isabell die mir ununterbrochen Kraft und Motivation verlieh um diese Arbeit durchzuführen und zu vollenden. Ich hoffe ich kann dir genau so viel Motivation und Kraft zurückgeben wenn du es so benötigst wie ich. Ich kann gar nicht zählen wie viele Stunden du mit Korrekturlesen verbracht hast.
- Meinen Freunden in Kärnten, Wien, Bayern und Anderswo, die immer ein offenes Ohr und Zeit für notwendige Ablenkung fanden. Rauchende Köpfe wurden fachgerecht gekühlt und stattdessen die Grillkohlen zum Glühen gebracht. Die so gemeinsam verbrachte Zeit ist mit absolut nichts aufzuwiegen.
- Meinen Kommilitonen und Mitbestreiter aus der Bürorunde. Ich habe nicht nur immer fachlich immer Rat und Hilfe erhalten wenn ich sie benötigte, auch die gemeinsam verbrachte Freizeit bereitete mir viel Freude. Selbst Diskussionen über "chicken"[1] brachten oft ersehnte Ablenkung.
- Meinen Betreuern Dr. Christian Krieg und Prof. Axel Jantsch für die Unterstützung bei der Durchführung dieser Arbeit.

Dankeschön

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[1] Douglas E. Zongker. “Chicken Chicken Chicken : Chicken Chicken”. In: https://isotropic.org/papers/chicken.pdf. 2002.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

We propose a method that is capable of performing *gate level information flow tracking (GLIFT)* using security policies expressed as a *pattern graph specification language (PGSL)* pattern. Using it we identify information flows violating a hardware design's specification. Sensitive data is often prohibited from being transferred to certain parts of a hardware design such as an output port. Such an information leak can be the result from different causes, such as implementation errors, timing based side channels or malicious information leaks introduced by *third-party IP (3PIP)*.

Different approaches have been published that try to conclude on whether there is an information leak or not. GLIFT as used by SecVerilog tracks the influence of input values to output values on the gate level whereas *register transfer level information flow tracking (RTLIFT)* tracks information flows on the *register transfer level (RTL)*. GLIFT adds so-called shadow logic to the design based on its initial gate level cells. This increases the complexity of the hardware design which is expressed as the number of cells and wire bits and increases the execution time of operations done on the design and formal verification. The state-of-the-art method SecVerilog uses SMT-LIB syntax combined with non-standard Verilog syntax. The policies used by SecVerilog are both complicated and can not be modified in a user-friendly way. In those works we identify the lack of a comprehensive description language that allows defining security policies that have high human-readability and are simple to create. We show that we can utilize the syntax of PGSL, a regex like graph search methodology, to create a security policy that represents a hierarchy of security domains. In order to identify information leaks based on this security policy we add information to objects in the design using Verilog attributes. Security labels are introduced to the *hardware description language (HDL)* using available Verilog syntax that does not interfere with any synthesis process done outside our method. We then use formal verification to identify possible information leaks violating the security policy. To reduce the resource consumption and the execution time of formal verification, caused by the additional design elements originating in shadow logic, we use knowledge of security domains and create segments of the design.

We provide a motivating example showing the usage of our method by testing it on a modified version of the hardware Trojan AES128-T100 provided by *trust-hub.org*. Then we summarize previous work and methods. We further compare our method to the state-of-the-art method SecVerilog. Our results show that our method uses a more human-readable security policy that also allows modifications with less overhead. This increases simplicity and makes our method more simple to use. Further, this potentially reduces human errors at creating security policies.

Using an exemplary design that features an unspecified information flow to an output we demonstrate how we compose a security policy using PGSL and show how our tool chain is used to detect this unspecified information flow. After that, we provide a modified version of this hardware design where the information leak is avoided using additional logic. In this design our method does not identify an information leak. These designs also present our benchmarks to validate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

our methodology. Further, we test our method on the designs AES-T100 and RSA-T100 provided by *trust-hub.org* and show that it is able to detect information flows.

We analyze the execution time overhead created by the segmentation process and perform experiments that confirm our analysis and assumptions. Both shadow logic creation and segmentation are of linear complexity $f_{\text{glift}} = f_{\text{segmentation}} = \mathcal{O}(n)$ with n being the number of cells and wires in the design. We measure an execution time overhead caused by the segmentation process with a maximum of 15.5 %. Verification time can be reduced based on the number of cells and wire bits that are removed at segmentation. We measure a reduction of 6.4 % in verification time when removing 2.1 % of wire bits and 0.06 % of cells in the AES128-T100 design. Overall the execution time of formal verification could be reduced by 1.9 % using segmentation based on the security policy.

We conclude that our method is capable of identifying unspecified information flow using GLIFT and formal verification. In the design AES128-T100 we could identify an information flow in 794 s which we could reduce to 743 s using our segmentation process. Information leaks are detected in all test designs that feature such an information flow.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

In dieser Arbeit schlagen wir eine Methode vor, die in der Lage ist Informationsflüsse auf Gatterebene zu verfolgen (GLIFT). Dadurch identifizieren wir Informationsflüsse, welche die Spezifikation eines Hardware-Designs verletzen. Ein solches Informationsleck kann zum Beispiel durch eine fehlerhafte Implementierung oder einen zeitbasierten Seitenkanal entstehen. Auch sind böswillige Änderungen im Entwurfsprozess oder bei der Verwendung von externen Bibliotheken (3PIP) möglich. Gängige Methoden wie GLIFT verfolgen den Einfluss von Information auf Gatterebene um ein Informationsleck aufzuspüren. In diesen Arbeiten fehlt es an einer leicht verständlichen Beschreibungssprache um Sicherheitsrichtlinien zu formulieren und den Entwurf dahingehend zu verifizieren. State-of-the-Art Methoden wie SecVerilog verwenden zum Beispiel die SMT-LIB Syntax in Kombination mit neu eingeführten Sprachkonstrukten. Wir nutzen anstelle der gängigen Methoden die Syntax der "Pattern Graph Specification Language (PGSL)", einer graphbasierten Suchmethode. Mit ihrer Hilfe erstellen wir eine Sicherheitsrichtlinie, welche eine Hierarchie von Sicherheitsdomänen repräsentiert wie sie beispielsweise vom Sicherheitsmodell Bell-LaPadula verwendet wird. Um Informationslecks auf der Grundlage der Sicherheitsrichtlinien zu identifizieren, fügen wir den Objekten in der Hardwarebeschreibungssprache (HDL) zusätzliche Informationen hinzu, um sie Sicherheitsdomänen zuordnen. Wir verwenden die verfügbare Verilog-Syntax, um den Syntheseprozess außerhalb unserer Methode nicht zu beeinträchtigen.

Anhand des Hardware-Trojaners AES128-T100 zeigen wir wie unsere Methode verwendet werden kann. Der Hardware-Trojaner wird dabei von *trust-hub.org* bereitgestellt. Anschließend fassen wir frühere Arbeiten und Methoden zusammen und präsentieren darauf aufbauend unsere Methode. Außerdem vergleichen wir die von uns entwickelte Methode mit der State-of-the-Art-Methode SecVerilog. Die Ergebnisse zeigen, dass unsere Methode flexibler ist, da die verwendeten Sicherheitsrichtlinien lediglich aus einer einzigen Codezeile bestehen. Anhand eines vereinfachten, beispielhaften Designs, welches einen nicht spezifizierten Informationsfluss zu einem Ausgang aufweist, demonstrieren wir wie wir eine Sicherheitsrichtlinie mit PGSL erstellen. Wir zeigen wie unsere Tools verwendet werden können, um diesen nicht spezifizierten Informationsfluss zu erkennen. Im Anschluss erstellen wir eine modifizierte Version dieses Hardware-Designs, welches der Spezifikation entspricht und somit kein unbeabsichtigter Informationsfluss mehr stattfindet. Wir zeigen nun, dass unsere Methode nun keinen Informationsfluss mehr erkennt der die Sicherheitsrichtlinie verletzt und die Methode somit funktioniert.

GLIFT fügt dem Hardware-Design Schattenlogik hinzu, die auf der Netzliste auf Gatterebene basiert. Dies erhöht die Komplexität des Designs, welche wir anhand der Anzahl der Zellen und Verbindungen auf Gatterebene messen. Um den Ressourcenverbrauch und die Ausführungszeit eines formalen Verifikationsprozesses zu reduzieren, nutzen wir das Wissen über Sicherheitsdomänen und erstellen Segmente des Designs. Wir analysieren den Anstieg der Ausführungszeit, der durch den Segmentierungsprozess entsteht und führen Experimente durch, die unsere Analysen und

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Annahmen bestätigen. Sowohl die Erstellung der Schattenlogik als auch die Segmentierung sind von linearer Komplexität $f_{\text{glift}} = f_{\text{segmentation}} = \mathcal{O}(n)$, wobei n der Anzahl der Zellen und Verbindungen im Design entspricht. Durch den Segmentierungsprozess reduzieren wir die Anzahl der Zellen und Verbindungen in dem Design, was sowohl den Speicherverbrauch als auch die Ausführungszeit von Syntheseprozessen reduziert. Die Segmentierung selbst erhöht jedoch die Ausführungszeit. Wir messen einen Anstieg der Ausführungszeit von maximal 15 %, welcher durch den Segmentierungsprozess verursacht wird. Die Verifikationszeit kann danach, je nach Anzahl der Zellen und Verbindungen die bei der Segmentierung entfernt werden, reduziert werden. Wir messen eine Reduzierung der Verifikationszeit um 5,7 %, wenn 2,1 % der Verbindungen und 0,06 % der Zellen im AES128-T100-Design entfernt werden. Durch diese Reduktion der Verifikationszeit können wir die gesamte Laufzeit unserer Methode um 1,9 % verkürzen.

Wir kommen zu dem Schluss, dass unsere Methode in der Lage ist nicht spezifizierte Informationsflüsse mit Hilfe von GLIFT und formaler Verifikation zu identifizieren. Unsere Methode entdeckt solche Informationslecks in allen gewählten Testdesigns die einen solchen Informationsfluss aufweisen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Contents

Abstract **5**

Kurzfassung **8**

1 Introduction **21**

1.1 Overview . . . 22

1.2 Problem statement . . . 23

1.3 Research questions . . . 23

1.4 Contributions . . . 23

2 Motivating example **25**

2.1 Problem description . . . 26

2.2 Security Domains . . . 26

3 Preliminaries **31**

3.1 Security threats in hardware . . . 31

3.2 Information security . . . 33

3.3 Information flow tracking . . . 40

3.4 PGSL . . . 49

3.5 Hardware verification . . . 53

3.6 Related work . . . 54

4 Methodology **57**

4.1 PGSL pattern as security policy . . . 58

4.2 GLIFT . . . 62

4.3 Formal verification . . . 62

4.4 Tool chain . . . 64

4.5 PGSLIFT example . . . 65

5 Implementation **73**

5.1 GLIFT . . . 75

5.2 PGSLIFT . . . 80

5.3 Formal verification . . . 87

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

6 Results **89**

6.1 Methods . . . 89

6.2 Test designs . . . 90

6.3 Information leak detection . . . 93

6.4 Complexity analysis . . . 94

6.5 GLIFT pass - shadow logic creation times . . . 103

6.6 PGSLIFT - tool execution times . . . 107

6.7 Formal verification times . . . 110

7 Discussion and Conclusions **113**

7.1 Discussion . . . 113

7.2 Conclusions . . . 117

Bibliography **121**

A Appendix **123**

A.1 Simple-bus . . . 123

A.2 Simple-bus-secure . . . 124

A.3 Tools . . . 125

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

3.1 Interpretation of taint values for an AND gate. 43
3.2 Complete truth table for the shadow logic function of an AND gate. Minterms are highlighted. 43
3.3 Gates for creating shadow logic for MUX gates 46
3.4 Structural language elements of *pattern graph specification language (PGSL)*. 52

4.1 PGSL elements used to describe a domain hierarchy. 58
4.2 Truth table of a 2:1 MUX with IFT extension. 67
4.3 Truth table for DEMUX with shadow logic 68
4.4 Truth tables with shadow logic for MUX and DEMUX. 68
4.5 Controlling the select inputs to mitigate the "insecure" state. 71

5.1 Lookup dictionaries created by the pass. 80

6.1 Test designs used in this thesis. 90
6.2 Elements of design simple-bus. 90
6.3 Elements of design simple-bus-secure. 91
6.4 Elements in the design AES128-T100 91
6.5 Comparison of cells and wires of design RSA-T100 in different states. 92
6.6 Elements in the design RSA-T100 93
6.7 Execution times for compiling and matching the regular expression and patterns. 99
6.8 Elements in the test design for performing memory consumption tests. 101
6.9 Chosen options for the verification processes. 103
6.10 GLIFT pass execution times for test designs 104
6.11 GLIFT pass execution times details. 105
6.12 GLIFT pass execution time for alternative test designs. 106
6.13 Summarized execution times for running PGSLIFT script. 107
6.14 Detailed execution times of running the PGSLIFT script. 109
6.15 Verification tool execution times for all test designs 110
6.16 Detailed execution times of running the verification script on our test designs. 111

7.1 Summary of ell and wire overhead when adding shadow logic to a design. 114
7.2 Comparing SecVerilog to *pattern graph specification language information flow tracking (PGSLIFT)*. 115

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 Process of applying PGSLIFT to a hardware design. 26
2.2 Wrapper around the DUT. 27
2.3 Top module of AES128-T100. 28
2.4 Netlist representation of the AES128 IP and the Trojan AES128-T100. 28

3.1 Graphical representation of the CIA triad. 34
3.2 Symbolic information flow in a system with multiple inputs and outputs. 36
3.3 Domain hierarchy for the document classification example. 38
3.4 Combining the security models of Bell-LaPadula and Biba with the domain example from Section 3.2. . . 39
3.5 Information flow in a logical function f with inputs A, B and output Y. 42
3.6 Shadow logic for an AND type gate. 44
3.7 Karnaugh map for precise and imprecise shadow logic of AND gates. 48
3.8 Implementation of the logic function of a MUX gate. 48
3.9 Using XOR gates for shadow logic of a MUX gate. 49
3.10 The grammar of *pattern graph specification language (PGSL)* in *Backus-Naur form (BNF)*. 51
3.11 Default security lattice used by SecVerilog. 55

4.1 Wrapper module connecting to inputs and outputs of the DUT 59
4.2 Combining shadow logic and the wrapped DUT. 62
4.3 We connect two modules directly to each other. Our created module has the same port specification. . . 63
4.4 PGSLIFT tool chain 64
4.5 Example design without IFT logic. A leak of the key to the UART-register can not be detected automatically. 65
4.6 Example design with IFT logic. 67
4.7 Example design enhanced with IFT logic and additional module to the control select inputs (yellow colored). 71

5.1 Tool chain developed to identify information flows using PGSL patterns. 74
5.2 Cloning flip-flop cells into shadow logic. 78
5.3 Creating shadow logic for two connected cells 78
5.4 Verification flow without segmentation. 83
5.5 Verification flow "segmentation first, *gate level information flow tracking (GLIFT)* later" — *SFGL*. 84
5.6 Verification flow "GLIFT first, segmentation later" — *GFSL*. 84
5.7 Verification process 87

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

6.1 PGSLIFT tool functions. 97
6.2 Measured execution times of (a) *delete* and (b) *clean* passes. 101
6.3 Memory consumption before and after segmentation 102
6.4 Measured execution times of GLIFT pass for our test designs 104
6.5 Measured execution times of GLIFT pass for smaller test designs 105

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listings

2.1 Original AES128 top module definition. 27
2.2 Wrapper module with defined security domains 27
2.3 PGSL pattern describing the domain hierarchy which reassembles our security policy. 29
2.4 SystemVerilog file that adds assertions and assumptions. 29
2.5 Created SymbiYosys configuration that allows to verify the design *AES128-T100*. 29
3.1 Short example of a *Java information flow (Jif)* class definition. 41
3.2 PGSL pattern used for detecting a *linear feedback shift register (LFSR)*. 50
3.3 MUX example provided by the authors of SecVerilog. 56
3.4 Function definition for a security label change in SecVerilog. 56
3.5 Security policy definition used in SecVerilog. 56
4.1 A PGSL pattern is used to describe a domain hierarchy. 58
4.2 *PGSLIFT* pattern that describes a policy. 58
4.3 The input In0 is marked with the security domain "Secure". 59
4.4 Module definition for the SVA module 63
4.5 Top module connecting the verification IP to the wrapper module. 63
4.6 Symbiyosys configuration example. 64
4.7 Definitions of security domains by using Verilog attributes 66
4.8 Loading and preparation of the design. The result is a single module at gate level. 69
4.9 PGSL pattern as proposed and used in this thesis to describe an unallowed information flow. 69
4.10 Assertion to finding information flows by monitoring the taint output. 69
4.11 Exemplary report of verifying the design simple-bus. 70
4.12 Exemplary report of verifying the design simple-bus-secure. 71
5.1 Changes applied to the Yosys Makefile that enable the support for Pyosys wrappers. 74
5.2 Basic structure to create a Yosys pass using Pyosys wrappers. 75
5.3 Regular expression used for parsing the PGSL pattern. 81
5.4 PGSL pattern elements split by blocks. 81
5.5 List to store domain hierarchy. 82
5.6 Segmenting the design by deleting outputs not part of the solution. 83
5.7 Assertion needed for formal verification. 84
5.8 SystemVerilog assertions used to verify the example design. 85
5.9 Directory tree for saving output files. 85
6.1 Security policy for identifying an information leak in test design AES128-T100. 91

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2 Preparation script for loading and synthesizing the RSA-T100 design . . . 92
6.3 Wrapper for test design RSA-T100. . . . 93
6.4 Exemplary report of verifying the test design RSA-T100. . . . 94
6.5 Exemplary report of verifying the test design AES128-T100. . . . 94
6.6 Preparation script to load test designs. . . . 95
6.7 Pattern generation for all test cases given a unsorted domain list. . . . 98
A.1 Wrapper that adds security domains to the top module. . . . 123
A.2 Top module connecting MUX and DEMUX. . . . 123
A.3 MUX module. . . . 124
A.4 DEMUX module. . . . 124
A.5 Secure top module that introduces the control module. . . . 124
A.6 Control module that controls the select inputs of MUX and DEMUX. . . . 124
A.7 We import libyosys from different locations depending on the __name__ variable. . . . 125

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Glossary

3PIP Third-party IP 5, 21, 31, 32

ACL Access control list 38

AES Advanced encryption standard 33, 35

AIG And-inverter graph 46, 80

ASIC Application-specific integrated circuit 47

AST Abstract syntax tree 49

BNF Backus-Naur form 13, 49–51

CAG Cell adjacency graph 49

CLI Command-line interface 85

CPU Central processing unit 34, 35, 83

DFA Differential fault analysis 36

DIFC Decentralized information flow control 41

DIFT Dynamic information flow tracking 40

DNF Disjunctive normal form 46

DoS Denial-of-service 33

DPA Differential power analysis 35

DUT Design under test 21, 25, 27, 40, 53, 59, 62, 63, 66, 69, 81, 85, 118

DUV Device under verification 53

FSM Finite-state machine 31

GLIFT Gate level information flow tracking 5, 6, 13, 22–25, 27, 42, 49, 54, 55, 57–59, 62, 73, 84, 86, 99, 107, 113, 117

HDL Hardware description language 5, 22, 25–27, 34, 53, 54, 75, 87, 116, 117

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

HT Hardware Trojan 22, 25, 26, 31–33, 50, 91, 92, 113

IC Integrated circuit 34–36

IFT Information flow tracking 21–23, 27, 29, 32, 33, 40–42, 47, 49, 57, 59, 61, 66, 117

IoT Internet of things 32, 33

IP Intellectual property 21, 31–33, 62, 70, 85

Jif Java information flow 15, 41

LFSR Linear feedback shift register 15, 50

NATO North atlantic treaty organization 33

NSA National security agency 33

NTFS New technology file system 38

PGSL Pattern graph specification language 5, 11, 13, 15, 22, 23, 25, 26, 29, 49–52, 57, 58, 60, 64, 69, 73, 74, 81, 86, 90, 97, 98, 113, 115–118

PGSLIFT Pattern graph specification language information flow tracking 11, 13, 14, 25, 26, 64, 70, 81, 86–88, 90, 92, 97, 102, 107, 110, 115, 116, 118

Pyosys Python wrappers for Yosys 22, 24, 54, 73, 74, 80, 92, 125

RSA Cryptographic method named after Rivest, Shamir, Adleman 35

RTL Register transfer level 5, 32, 33, 49, 117

RTLIFT Register transfer level information flow tracking 5, 49, 117

RTLIL Register transfer level intermediate language 54, 75, 77, 79, 80, 85, 95, 102, 126

SAT Satisfiability 102, 103, 116, 117

SELinux Security enhanced Linux 41

SIFT Static information flow tracking 40

SMT Satisfiability modulo theories 103

SoC System on chip 21, 31, 32

SPA Simple power analysis 35

SPI Serial peripheral interface 65

SVA SystemVerilog assertion 62, 84

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

SymbiYosys Formal verification frontend for Yosys synthesis tool 15, 22, 23, 29, 62, 64, 69, 70, 73, 80, 84–88, 90, 92, 103, 116

SystemVerilog An extension of Verilog HDL that allows formal verification 15, 22, 25, 29, 30, 53, 54, 62, 69, 84–86, 117

UART Universal asynchronous receiver transmitter 65

Verilog A HDL widely used in industry and research. Multiple standards were defined. 5, 27, 40, 50, 53, 54, 57–59, 62, 66, 70, 73, 74, 79, 82, 85, 92, 107, 115–117

VHDL Very high speed integrated circuit hardware description language 54, 92, 95

Yosys Open-source synthesis tool 22, 24, 27, 42, 49, 54, 61, 62, 64, 68–70, 73–75, 78–80, 83, 85–87, 92, 95, 99–101, 103, 107, 125

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 1

Introduction

The increased presence of digital circuits in our everyday life raises concerns about security of the data in electronic devices and the safety generated or insured by those devices. More and more attention is drawn on this topic as details on sensitive data leaks and manipulations emerge. Timing and cache side channels as well as other flaws in hardware designs can expose sensitive data to outsiders. Modern *systems on chips (SoCs)* are designed using pre-built design parts, commonly called *intellectual property (IP)* cores or *third-party IP (3PIP)* cores, often bought from external companies. The *system on chip (SoC)* designer often has no control or knowledge about the implementations of those parts.[1,2] So-called side channels can leak data through different logical or physical effects that can be exploited. Recent history shows that billions of devices suffer from built-in side channels that were not discovered until billions of processors with this flaw had been sold.[3,4]

The goal of this thesis is to create a comprehensive way of creating security policies that can be used in an automated way so that a given hardware design can be verified against them. One existing method to conquer this leakage of data is *information flow tracking (IFT)*. A lot of work in this field has been published by Ryan Kastner, Wei Hu and Armaiti Ardeshiricham in various works. It is a method to track the influence of data, rather than the actual values, across a hardware design.[5,6] So-called shadow logic, that is built based on the *design under test (DUT)*, implements a logic function that tracks this influence across the design. This shadow logic features so-called taint inputs and outputs. Values of taint inputs are used to control which information shall be tracked while taint outputs represent the presence of an information flow. Using formal verification, flows of information can be detected by monitoring the output value of this shadow logic. This method allows determining if information is transferred between an input and an output of interest.

While there are currently various methods to introduce this logic on different levels of abstraction to the hardware design, we identify a lack of a comprehensive way to describe security policies that shall be ensured. Available methods

1 Prabhat Mishra, Swarup Bhunia, and Mohammad Tehranipoor. *Hardware IP security and trust.* eng. Cham: Springer, 2017.
2 Swarup Bhunia et al. "Hardware Trojan Attacks: Threat Analysis and Countermeasures". In: *Proceedings of the IEEE* 102.8 (Aug. 2014).
3 P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2019.
4 Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". en. In: 2018. (Visited on 03/11/2021).
5 Mohit Tiwari et al. "Complete Information Flow Tracking from the Gates Up". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XIV. Washington, DC, USA: ACM, 2009.
6 Wei Hu, Ardeshiricham Armaiti, and Ryan Kastner. "Hardware Information Flow Tracking". In: (2020).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

introduce new description languages.[7] This requires verification engineers to have knowledge of this language. Further the available methods model information flow in the design using functional statements. We show that the lines of code needed for adding information as required by the method SecVerilog is high compared to the lines in the *hardware description language (HDL)*. We take the opposite approach and identify already available description methods used in hardware design and hardware verification. Therefore, we propose the use of the description language PGSL. It is already used in the field of hardware verification.[8] PGSL was initially created to identify malicious parts of a hardware design by describing these malicious elements in an abstract way. Its universal way of describing hardware parts allows us to describe a hierarchy of security domains that form our security policies used for verification. Using this language we manage to create security policies in a single line of code.

1.1 Overview

Chapter 2 shows a motivating example where we demonstrate the usage of our method by identifying information leaks in an cryptographic core that reassemble a *hardware Trojan (HT)*. We summarize threat models, state-of-the-art methods, IFT principles and related work in Chapter 3 Preliminaries. We introduce security domains, existing models that deal with information exchange between them. Our chosen description language PGSL is presented with a short usage example.

Based on the problem statement and the related work, we introduce our method and formulate research questions. Our method is capable of automatically applying GLIFT to a hardware design by adding security domains and a security pattern describing their hierarchy. Information flows are detected using state-of-the-art formal verification methods. We develop a verification flow that uses SystemVerilog assertions to analyze all possible information flows .[9] It is described in detail in Chapter 4 Methodology.

To demonstrate the usage of our method, we provide an example design consisting of an informal specification and an implementation. It contains a design flaw that leaks information to an output in an unspecified way. We show how our method is able to discover this flaw. We further provide a correction to this flaw and verify the design again in order to prove the formal correctness. This example is presented in Section 4.5 PGSLIFT example.

To create our tools we use the open source synthesis tool Yosys to modify hardware designs and add shadow logic to track information flows.[10] Using SymbiYosys, a formal verification frontend to Yosys, we perform formal verification on this modified hardware design.[11] The result of this verification indicates whether information flows to the outputs of interest or not. We implement our tools using Python, the open source synthesis tool Yosys which is used in modern hardware design flows and Pyosys.[12] We provide details on the implementation of our method in Chapter 5 Implementation.

A common way to benchmark IFT techniques is to measure the execution time of the formal verification.[13] We therefore provide timing measurements to compare our method to previous works. Our research questions focus on the changes in

[7] Danfeng Zhang et al. "A hardware design language for timing-sensitive information-flow security". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015).
[8] Christian Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Feb. 2019.
[9] Ashok B. Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2020.
[10] Claire Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.
[11] Clifford Wolf. *SymbiYosys*. https://github.com/YosysHQ/SymbiYosys.
[12] Benedikt Tutzer et al. "Python Wraps Yosys for Rapid Open-Source EDA Application Development". In: 2019.
[13] W. Hu et al. "Theoretical Fundamentals of Gate Level Information Flow Tracking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

execution time of formal verification when changing the hardware design using segmentation techniques. Segmenting the design also adds overhead in execution time and resource consumption. We measure this overhead and compare two different segmentation approaches. Results are presented in Chapter 6 Results.

1.2 Problem statement

We identify the absence of a comprehensive description language that allows to model simple and easy to understand security policies that can then be used for applying IFT techniques on a hardware design. State-of-the-art methods create security policies using the SMT-LIB syntax to model an abstract security lattice that is hard to examine without having deep knowledge of the syntax and their methodology. Shadow logic creation, which is needed to perform GLIFT, adds cells and wires to the design and therefore increases the resource consumption of the design in the verification environment. To reduce verification time and resource consumption in terms of memory usage we want to use knowledge about security domains to partition the design prior to creating shadow logic and verifying it.

1.3 Research questions

With this thesis and the help of our created method we find answers to the following research questions:

(1) Can we use PGSL to create simple and flexible security policies?

(2) How does creating design subsets using PGSL security policies influence verification time?

  (a) What is the overhead when creating subsets of the design?

  (b) What is the execution time when verifying a subset of the design?

1.4 Contributions

We use PGSL in a new way and demonstrate its versatility in order to create simple and easy to understand security policies. We detect information leakage by partitioning inputs and outputs of a hardware design in security domains. Using a PGSL pattern we define a hierarchy of security domains that we use as a security policy. It describes valid information flows between distinct security domains. By using formal verification we identify information flows that violate this policy and create an easy-to-read report. Our created tool chain is capable of verifying all test designs that we analyzed. We detect information leaks in all designs that feature such.

Scientific contributions

Our goal is to find a way to use abstract description language PGSL to perform formal verification and identify unspecified information flows. The method involves the creation of a hierarchy of security domains that forms the security policy for verification. We research how the execution time of shadow logic creation as needed for GLIFT can be influenced by altering the number of cells and wires in the design. Our results show that the execution time of adding shadow logic to perform GLIFT is linear with the number of cells in the design. That means we can reduce the execution time by segmenting the design based on the security domains. We further research the relation between execution time of formal verification tool SymbiYosys and the complexity of a design in terms of netlist elements. Our measurements

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

show the accuracy of these assumptions. Reducing the number of cells in a design by 0.58 % and wire bits by 14.4 % using segmentation lead to an execution time reduction of a formal verification attempt by 6.42 %. A complexity analysis of our methods further shows how execution times can be reduced by reducing the amount of netlist elements (cells and wires).

Technical contributions

The created GLIFT plugin for Yosys will be published under an open source license which will allow engineers to use the implementation and modify the techniques. We decided to use Python as the programming language which allows for more versatility in modifications in future works. This will hopefully lead to a wider distribution of the presented methods so researchers and engineers can adopt the presented methods at their own needs.

Using open source software to both create our tools, synthesize our chosen test designs and to verify our modified designs allows future researchers to enhance our method. This also boosts the overall usage of these tools and builds the base for future development. Further, the development of our tools resulted in some issues being found and reported in the used software library Pyosys, some of them have already been fixed. This work also lead to development in the synthesis tool Yosys regarding better integration of Pyosys in future releases.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 2

Motivating example

Pattern graph specification language information flow tracking (PGSLIFT) is a method to identify security violations and issues in a chip design by tracking read and write access between predefined security domains. We combine *gate level information flow tracking (GLIFT)* and formal verification to verify the design against a security policy expressed as a *pattern graph specification language (PGSL)* pattern that describe a hierarchy of security domains. They can be defined in the *hardware description language (HDL)* specification of the design. Therefore, the HDL code of the *design under test (DUT)* needs to be available.

In this chapter we present a real world use case for our proposed method and show how a verification engineer creates a security policy. We demonstrate this by using the implementation of an AES128 core provided by Trust-Hub.[1][2] This design reassembles a *hardware Trojan (HT)* that leaks unencrypted data to an additional output when a trigger condition is true.

Figure 2.1 illustrates the verification process. Information about security domains are added to the HDL source files. The design is then synthesized to a gate level netlist and shadow logic is added that enables us performing GLIFT. A PGSL pattern is introduced as a security policy. Based on it, we remove inputs and outputs not necessary for the verification process to create segments of the original design that can be verified independently. SystemVerilog assertions are added that allow for formal verification. After running the verification tool we investigate the result. If the verification passes no information flow could be detected. In case an information leak is detected the input state leading to the failed verification is extracted from the verification tools output files. At the end a report is created containing the outputs of the verification tool and information about the detected information leak.

[1] H. Salmani, M. Tehranipoor, and R. Karri. "On design vulnerability analysis and trust benchmarks development". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. Oct. 2013.

[2] Bicky Shakya et al. "Benchmarking of Hardware Trojans and Maliciously Affected Circuits". In: *Journal of Hardware and Systems Security* 1.1 (Mar. 2017).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

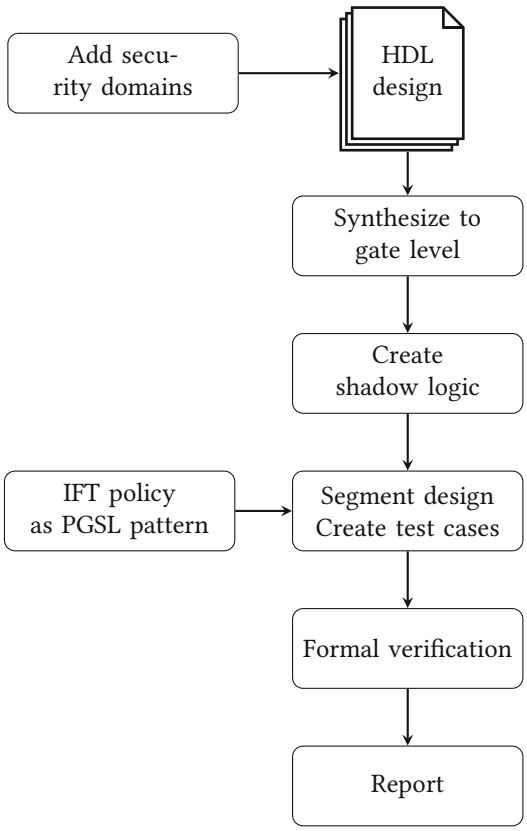


Figure 2.1: The process of applying PGSLIFT to a hardware design. The verification engineer needs to provide the security domains and the PGSL pattern describing the domain hierarchy.

2.1 Problem description

When using a design from a third-party security concerns arise immediately. Especially when using designs that implement a security-critical function, such as cryptographic, it is necessary to check the design for design flaws or for containing malicious parts (such as backdoors, HT, etc.). In most cases the implementation itself can not be formally verified and therefore, we need methodologies to ensure trustworthy high-level HDL descriptions. The approach presented in this thesis focuses on finding potential security vulnerabilities, in particular information leakage, by evaluating the design as a gate level netlist. The verification engineer introduces security domains to the design. By defining a hierarchy of those domains, he introduces a security policy which we will be able to verify the design against. This hierarchy is formalized using PGSL, a language already present in modern hardware verification. Knowledge of the security domain hierarchy allows us to reduce the complexity of the hardware design by creating test cases where each test case represents a possible information flow from one to another security domain. We make use of this reduced design complexity to speed up the formal verification process.

2.2 Security Domains

Security domains are introduced by creating a wrapper module around the original design. Attributes of wires in the wrapper propagate through the design. The wrapper allows to assign all inputs and outputs to a security domain by using attributes in the declaration of wires. This way, the given hardware design is not altered. Alternatively, information about the security domains can be added to the original design since they do not affect the synthesis process.[3]

[3] Stuart Sutherland. *Verilog® HDL*. 2001.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The top module definition of the AES128 design is shown in Listing 2.1. Listing 2.2 demonstrates how a wrapper can be created that assigns an input or output of the design to a security domain. We illustrate such a wrapper in Figure 2.2.

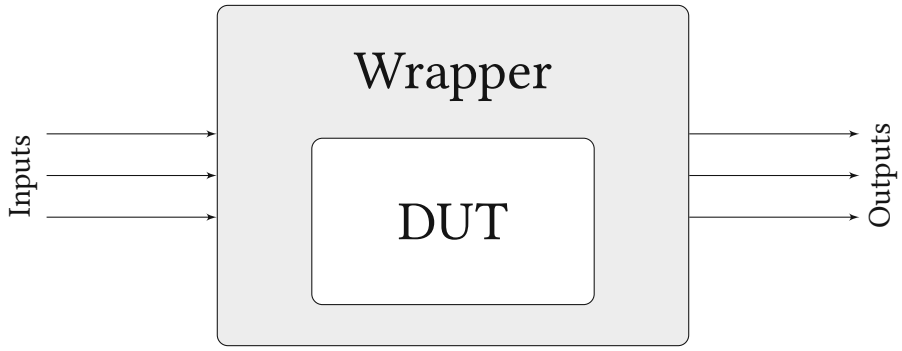


Figure 2.2: The wrapper connects to all inputs and outputs of the DUT. It has the same port definition as the DUT and connects directly to it. At this port definition we can add Verilog attributes that assign a security domain to each port.

Each input in the design gets assigned a security domain. Its name can be chosen without any restriction. We therefore create security domains that match the name of the corresponding port of the design.

```
module aes_128(clk, state, key, out);

    input          clk;
    input  [127:0] state, key;
    output [127:0] out;
    reg    [127:0] s0, k0;
    wire   [127:0] s1, s2, s3, s4, s5, s6, s7, s8, s9,
                   k1, k2, k3, k4, k5, k6, k7, k8, k9,
                   k0b, k1b, k2b, k3b, k4b, k5b, k6b, k7b, k8b, k9b;
```

Listing 2.1: Original AES128 top module definition as provided on *trust-hub.org*.

```
module aes_128_wrapper(clk, state, key, out);
    (*sec_domain="CLK"*) input clk;
    (*sec_domain="DATA"*) input  [127:0] state;
    (*sec_domain="KEY"*) input  [127:0] key;
    (*sec_domain="OUT"*) output [127:0] out;

    aes_128 top (clk, state, key, out);

endmodule
```

Listing 2.2: Wrapper module with defined security domains

Security policy violations

We synthesize the HDL into a gate level netlist using the synthesis tool Yosys .[4] A security issue is a violation of the rules described by the security models BIBA[5] and Bell-LaPadula.[6] Such violations can be found using *gate level information flow tracking (GLIFT)* which is a gate level implementation of *information flow tracking (IFT)*.

The design AES128-T100 as provided by Trust-Hub.org[7] is a AES128 core with a known security issue that has a malicious part in it that leaks the key to a distinct output called *Capacitance*. In its original state the design is not synthesizable to

[4] Wolf, *Yosys Open SYnthesis Suite*.
[5] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. en. Tech. rep. Apr. 1977. Chap. Technical Reports. (Visited on 03/04/2021).
[6] Bell, D Elliott and LaPadula, Leonard J. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.
[7] Trust-Hub.org. *Trust-Hub*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

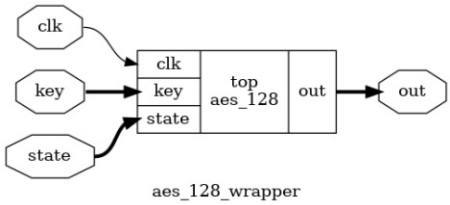


Figure 2.3: Netlist representation of the AES128-T100 top module. The malicious part can not be identified in this representation.

gate level. We therefore use a modified version of this design that leaks the key to the normal output of the encryption core and is also synthesizable. Figure 2.3 shows the top module in a netlist representation. The verification engineer can not identify that the key leaks to the output in an unintended manner.

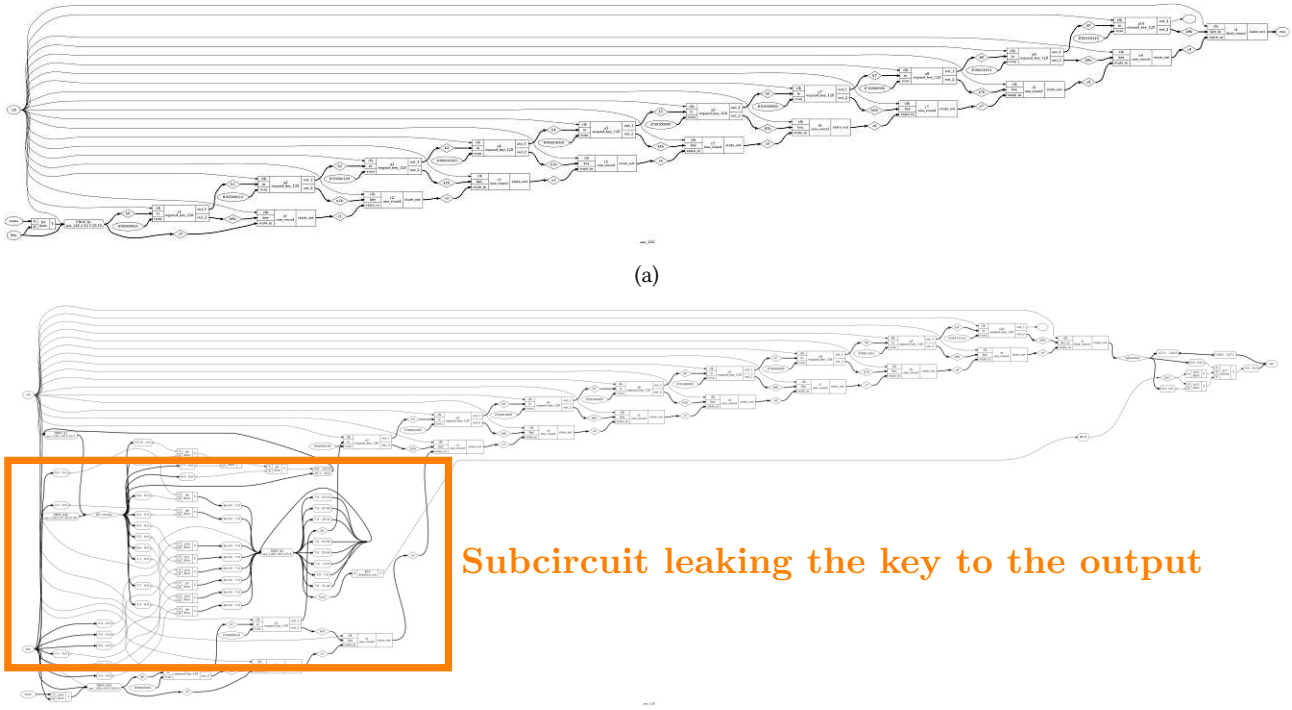


Figure 2.4: Netlist representation of the IP AES128 shown in (a) and the Trojan AES128-T100 shown in (b). Here, with the knowledge of the presence of a Trojan it can be identified as the lower part of the netlist where the key is transferred to the output. The malicious part of the design is highlighted in a red box.

Figure 2.4 shows the design both with and without the malicious module. Here the Trojan could eventually be identified by a skilled verification engineer that has knowledge of the Trojan free version. The malicious parts of the design are located on the bottom of the netlist where the key is transferred to the output. We highlighted the malicious part with a red rectangle.

Domain hierarchy

To identify information flow that violates the design's specification we create a hierarchy of domains. In this example, we are only interested in information being transferred from domain KEY to OUT. Our security policy can therefore be expressed as shown in Listing 2.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
P=KEY->OUT
```

Listing 2.3: PGSL pattern describing the domain hierarchy which reassembles our security policy.

This hierarchy is analyzed and processed by our tool. It creates a test case consisting of a segmented design and its source files, a SystemVerilog file and a configuration file for formal verification. Listing 2.4 shows the SystemVerilog file. Assumptions are added for inputs of IFT shadow logic and assertions for shadow logic outputs.

```
module AES_T100_tb(
  input clk,
  input clk_t,
  input [127:0] state,
  input [127:0] state_t,
  input [127:0] key,
  input [127:0] key_t,
  output [127:0] out,
  output [127:0] out_t
  );

  AES_T100 DUT (clk,clk_t,state,state_t,key,key_t,out,out_t);

`ifdef FORMAL
  always @(*) begin
    assume(clk_t == 1'b0);
    assume(state_t == 128'b0);
    assume(key_t == 128'b1);
    assert(out_t == 128'b0);
  end
`endif

endmodule
```

Listing 2.4: Created SystemVerilog file that instantiates a top module and add assertions and assumptions needed for identifying information leaks.

Formal verification

Formal verification is done using SymbiYosys. Our tool creates a configuration file that allows to perform the verification process. This file is shown in Listing 2.5. It is responsible for providing information on the design and preparation steps as well as options that control the verification process. We describe its content in detail in Section 5.3.

```
[options]
mode bmc
depth 5
expect pass

[engines]
smtbmc

[script]
read_ilang AES_T100.il
read_verilog -formal AES_T100.sv
hierarchy -check -top AES_T100_tb
proc
memory
clean
flatten

[files]
AES_T100.il
AES_T100.sv
```

Listing 2.5: Created SymbiYosys configuration that allows to verify the design *AES128-T100*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

According to Figure 2.1, formal verification is performed, and then the result is evaluated. If an unspecified information flow is detected from domain KEY to OUT, the verification process quits and produces a counterexample containing the set of inputs that lead to the failed verification attempt. Our tool analyzes this counterexample and adds a clause to the SystemVerilog file, so this set of inputs will be excluded in the next verification run. It then reports all found input value states that lead to an information flow. This way an information flow (also an information leak) is detected and reported.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 3

Preliminaries

With increasing presence of digital circuits in human society, concerns about safety and security of data in electronic devices get more and more attention as details on data leaks and manipulations emerge. Modern *system on chip (SoC)* are designed using prebuilt design parts, commonly called *intellectual property (IP)* cores or *third-party IP (3PIP)*, bought from external companies. The SoC designer has often no control or knowledge about the implementations of those parts.

Several security risks arise when using 3PIP as has been analyzed by Bhunia et al.:[1]

> "The possibility of Trojan attacks in third-party IP poses a major integrity concern to SoC designers. Verification of trust of an IP acquired from untrusted third-party sources can be extremely challenging due to lack of golden models.[2]"

This work focuses on the flow of information inside a hardware design in order to provide a proof of data security according to the specifications of a system. We will introduce the concept of information flows to identify such security threats in a hardware design. By also introducing abstract security policies and security domains we enable the system designer to verify the design at a higher level of abstraction compared to currently available methods.

3.1 Security threats in hardware

Mishra, Bhunia, and Tehranipoor[3] present an example on how a modern SoC is developed using various 3PIPs. They introduce a communication module IP with malicious behavior. This IP includes a backdoor, that allows access to its internal data and states which is a left over from design phase used for debugging. It further includes a HT, a part of the design that alters functionality in a malicious way. Hardware that treats data in such a malicious or unsafe way is considered a security threat. Using a construct called information flows enables us to identify such threats in a hardware design. Security threats can be segmented into intended and unintended information leaks by their creation.

Unintended information leaks result in errors or mistakes at the design stage of an IP. An example could be the existence of a debug state in a *finite-state machine (FSM)* that was not deleted before providing the design. Another example is a

[1] Bhunia et al., "Hardware Trojan Attacks".
[2] Ibid.
[3] Mishra, Bhunia, and Tehranipoor, *Hardware IP security and trust*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

cryptographic module that transports the key to its output when it gets initialized due to an error in the implementation. The field of hardware verification aims at verifying that the design meets its specification.[4] Two commonly used methods are simulation and formal verification.

Intended information flows transfer information such as cryptographic keys or unencrypted data. One type of such a threat is the *backdoor*. They are inserted on purpose to allow those aware of it to access the system in order to modify its behavior or data. A manufacturer could use such a backdoor to access a device in the field to provide updates or read log files. Also, a company could use the backdoor to access user data. No matter what the designer's intention was, backdoors present a security risk in every scenario.

A backdoor, even when implemented in noble intention, could potentially be discovered by an external individual and be used to harm the system. Malicious function that is able to manipulate the function of the design itself. Such parts of a design are commonly called HT as will be discussed later. These malicious parts are not operable the whole time but need an activation which differentiates it from a backdoor.

Krieg, Wolf, and Jantsch[5] describe a HT as:

> "A hardware Trojan is an electronic system that serves a shadow purpose besides its intended functionality. The hidden functionality is unspecified and undocumented, and included in a malicious manner."[6]

This list of threats is not complete by any means. There are several other security threats, but they all share the same intention in leaking or manipulating data. Mishra, Bhunia, and Tehranipoor[7] provide an overview on modern security and trust issues. They focus on the design and verification stage of a SoC in detail. Different attack models and threat models are discovered and analyzed. Further, they discuss and evaluate multiple verification methods for security purposes. Possible threats when using 3PIP are split in three domains:

- Malicious structural modifications. This is a typical hardware Trojan that modifies the behavior after being triggered.
- Hidden backdoors (due to test/debug interface) are a remainder of the design process and have not been removed appropriately.
- Malicious parametric changes such as power and performance attacks or reliability attacks.

The analyzed so-called soft IPs which is a *register transfer level (RTL)* design, firm IPs being a gate level netlist and hard IPs being a layout level design. All variants of IP are vulnerable as shown by their research. They identify two main threat models: untrusted IP providers and unspecified IP behavior. Connecting those models to potential data leaks and therefore information leaks we identify both threat models relevant for this work. Using our method allows us to identify malicious structural modifications and hidden backdoors by applying IFT to a soft or firm IP. Hard IP are outside the scope of this work we rely on analyzing the design at gate level or above.

Venugopalan and Patterson[8] discuss the threat of HT in the context of *Internet of things (IoT)* environments. They provide

[4] Mehta, *SystemVerilog Assertions and Functional Coverage*.
[5] Christian Krieg, Clifford Wolf, and Axel Jantsch. "Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow". In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD '16. Austin, Texas: ACM, 2016.
[6] Ibid.
[7] Mishra, Bhunia, and Tehranipoor, *Hardware IP security and trust*.
[8] Vivek Venugopalan and Cameron D. Patterson. "Surveying the Hardware Trojan Threat Landscape for the Internet-of-Things". en. In: *Journal*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

possible attacks and how they could be prevented as well as state-of-the-art research. Using a so-called *IoT pyramid of pain* they show the impact of a HT in an IoT environment. They focus on threats of information leaks, *denial-of-service (DoS)* and functionality or specification modification. Given a complete specification and a golden model they are able to identify all HT effects at RTL using formal verification. Our method does not cover all of these threats as IFT solely aims at identifying information flows. The threat of information leaks is however covered by our method.

A recent example for trust issues in hardware systems that one can not verify to a satisfying point oneself, is the mistrust between *north atlantic treaty organization (NATO)* and a far eastern manufacturer of communication hardware.[9] Another example for trust issues with unverifiable hardware is the cryptographic chip called "Clipper". It was published and designed by the *national security agency (NSA)* and advertised as a secure cryptographic chip for telecommunication systems. However, it turned out to contain a backdoor to leak unencrypted data.[10]

Other recent examples for side-channel attacks are vulnerabilities found in modern processor architectures named *Meltdown*[11] and *Spectre*.[12] Meltdown allows accessing kernel memory from user space processes. Spectre enables applications to read other application's memory via a side-channel. Trust-Hub.org[13] provides a database of IP cores, as used in modern hardware design processes. They provide descriptions and implementations of IP that include malicious parts in order to retrieve information such as the key used in a cryptographic core. From this database we use the design *AES128-T100* which is a cryptographic core implementing the *advanced encryption standard (AES)* in its 128-bit key variant. It reassembles a Trojan that leaks data through a leakage circuit attached to the design. This test design is also part of the work of Nahiyan et al.[14]

3.2 Information security

The so-called CIA triad[15] defines three objectives of information security: confidentiality, integrity and availability. Together, these three fundamental properties form a system, therefore they are usually described with a triangle. They can be applied to any system that deals with information security. When designing such a system with context in data handling and security, the system designer should always think of all three properties.

The term *design* will be used frequently in this chapter and overall describes a hardware design without being restricted to a specific form of representation.

Figure 3.1 illustrates these properties in a graphic way.[16]

- **Confidentiality** describes the protection of information from unauthorized access. In general, this objective is most important when dealing with cryptographic in order to encrypt or decrypt data such that only those who have permission (i.e., hold a copy of the secret key that is needed for the cryptographic process), can interpret data.

of Hardware and Systems Security 2.2 (June 2018). (Visited on 03/11/2021).

9 Kadri Kaska, Henrik Beckvard, and Tomas Minarik. "Huawei, 5G and China as a security threat". In: *NATO Cooperative Cyber Defence Center for Excellence (CCDCOE)* 28 (2019).

10 Howard S. Dakoff. "The clipper chip proposal: Deciphering the unfounded fears that are wrongfully derailing its implementation". In: *J. marshall L. rev.* 29 (1995).

11 Lipp et al., "Meltdown".

12 Kocher et al., "Spectre Attacks".

13 Trust-Hub.org, *Trust-Hub*.

14 A. Nahiyan et al. "Hardware trojan detection through information flow security verification". In: *2017 IEEE International Test Conference (ITC)*. Oct. 2017.

15 A. J. Neumann, N. Statland, and R. D. Webb. "Post Processing Audit Tools and Techniques". In: *Proceedings of the NBS Workshop, Miami Beach, Florida, USA*. 1977.

16 Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

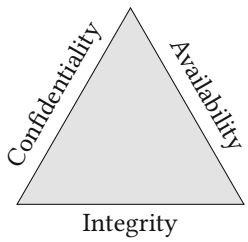


Figure 3.1: Graphical representation of the CIA triad.

Breaking confidentiality can, for example, be done by gaining access to the key in any possible way. Furthermore, confidential information can be leaked through so-called side-channels. Analysis of various physical properties can also leak information as seen in power analysis.[17][18]

- **Integrity** describes a system where information is not altered. This could happen when data is corrupted by an entity that is not in the same security domain as the data of interest. Integrity has high importance when accuracy of information is critical, for example in payment systems when dealing with account balances.

- **Availability** is the presence of the correct output value at any point in time. A system that provides crucial data needs to do this at a constant rate and can tolerate downtimes only to a certain amount. Availability is also important in the field of reliability engineering.

The CIA security objectives enable an engineer to focus on different aspects of security in the system. When designing a new system, all of these security objectives must be kept in mind and various steps can be taken in order to achieve security. A hardware design process starts with a specification (that is formulated as requirements in natural language) which is then implemented using a HDL. The functionality of the design is given by its *logic function*. The way how data is transferred in the system by its logic function influences both confidentiality and integrity. Availability is added by the physical implementation and its electrical and mechanical properties assuming that the logic function has no errors.

This work solely focuses on the properties confidentiality and integrity. Availability in all of its aspects is not verifiable by the used method *information flow tracking*, which we will elaborate in Section 3.3.

Side-channels

Depending on their nature, attacks that aim at extracting confidential information from a system can be called *side-channel attacks*. Here, confidential information is not leaked by the main functionality, but extracted from an effect of the product itself. For example, how the *integrated circuit (IC)* radiates electromagnetic information or how a cache in a *central processing unit (CPU)* operates. We can demonstrate the nature of side-channels by looking at password retrieval methods. When a password is retrieved by exploiting an error in the cache timing of a processor, we call it side-channel. However, when the same password is retrieved by using brute force methods or social engineering, it is not a side-channel. Calculating a cryptographic key via a *brute force attack* is not a side channel attack since it brakes the cryptographic system via trial and error without exploiting a side-channel. Hong[19] provides an overview on different

[17] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Annual international cryptology conference*. Springer, 1999.
[18] Daniel Genkin et al. "ECDSA key extraction from mobile devices via nonintrusive physical side channels". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016.
[19] Seokhie Hong. *Side Channel Attacks*. MDPI-Multidisciplinary Digital Publishing Institute, 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

types of side-channel-attacks. They are typically differentiated by their nature and the physical or logical effect they exploit:

- **Cache attacks** provide access to a shared cache memory that is shared between the victim and the attacker. Depending on the implementation, several attack vectors exist. For example, the CPU cache can be read by applications without being granted permissions due to vulnerabilities in look ahead systems. Two attacks that exploit vulnerabilities that were found in the last decade are Meltdown[20] and Spectre.[21]

- **Timing attacks** use the timing information of an operation to gather data. An example is the extraction of an AES or *Rivest-Shamir-Adleman(RSA)* key by measuring the computation time of the cryptographic function. The implementation of this function needs to be known beforehand. Differences in execution times are a result of the time delay originating from different lengths of data paths in the IC depending on the data involved in the computation. For example, the execution time of a *XOR* operation depends on its input variables and this fact allows reconstructing the inputs by measuring the timing of the output of the *XOR* gate.[22]

- **Power monitoring attacks** allow extracting of information from a system by monitoring its power consumption, in order to retrieve information via *simple power analysis (SPA)* or *differential power analysis (DPA)* as described by Kocher, Jaffe, and Jun.[23] A simple RSA implementation has different power consumption when multiplying bits with "0" or "1", which an attacker can use to calculate the key used for encryption. The power consumption of the chip is measured and matched with known data. This information is used to calculate the key, used in the encryption process.

- **Electromagnetic radiation** of a cryptographic device can be used to extract confidential information as shown by Genkin et al.[24] They use radiation generated by hardware during encryption to gather information on the key. Radiation is emitted depending on the functionality of the IC under operation. High frequency currents, which originate from the functionality of the IC, create radiation spreading across the fabric. This also relates to discrete circuits, where sharp edges occur on wires and discrete elements. The radiation of an electrical circuit depends on the design, layout, dimensions and the used frequencies.

- **Acoustic cryptanalysis** exploits sound waves created by cryptographic machines. An example is the sound created by a mechanical keyboard. It can be used to retrieve a key, similar to a keylogger but exploiting small acoustic differences of involved keystrokes. Also, it has been found that the power consumption in a processor can lead to vibrations caused by heating and cooling of the fabric creating soundwaves in a frequency range that can be measured acoustically. Genkin, Shamir, and Tromer[25] provide such a method and use it to extract the secret key used in a cryptographic process.

  > "The attack can extract full 4096-bit RSA decryption keys from laptop computers (of various models), within an hour, using the sound generated by the computer during the decryption of some chosen ciphertexts."[26]

[20] Lipp et al., "Meltdown".
[21] Kocher et al., "Spectre Attacks".
[22] k0resh. *Timing attacks - part 1.* en. Dec. 2013. (Visited on 05/27/2021).
[23] Kocher, Jaffe, and Jun, "Differential power analysis".
[24] Genkin et al., "ECDSA key extraction from mobile devices via nonintrusive physical side channels".
[25] Daniel Genkin, Adi Shamir, and Eran Tromer. "Acoustic Cryptanalysis". en. In: *Journal of Cryptology* 30.2 (Apr. 2017). (Visited on 05/19/2021).
[26] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- *Differential fault analysis (DFA)* is a subset of side-channel attacks where a fault is introduced to a system in order to gather information on internal states of the attack's victim. Fault injection could be done via temperature, voltage or current outside the specification. Furthermore, extreme overclocking or the exposition to light, radiation or strong electromagnetic fields are used to inject a fault. In addition, faults can be introduced on the data layer by providing data outside the designs' specification. These attacks are often named fault injection and methods such as voltage, electromagnetic, or light attacks exist.[27]

- **Data remanence** reads data after it should have been deleted before. An example is a simple hard drive. Depending on the file system, deleted data is usually not overwritten on the disk itself. Instead, the reference to it is deleted, so that the file system does not know that there is data on the disk anymore. Simple data recovery tools can be used to recover this data.

- **Optical** attacks use high resolution images. For example an image of a mask used to produce ICs, can be used to reconstruct the inner functionality of the chip. Furthermore, photons emitted by electrons changing states in transistors (which an IC is built upon). Knowledge of these state changes can be used to reconstruct keys in cryptographic functions as shown by Ferrigno and Hlavac.[28] They use widely available hardware to perform their *picosecond imaging circuit analysis.*

This list is by far not complete and only suited to provide examples of existing methods to exploit side-channels.

Information flows

Information flow is the propagation of information through a system or process. We illustrate this as shown in Figure 3.2. Here, we show how information is transferred from input A to output D. We can also express this as "information flows from *A* to *D*".

Lemma 1 *Information flows from input* A *to output* D *if* D *has knowledge of the information originating in* A.

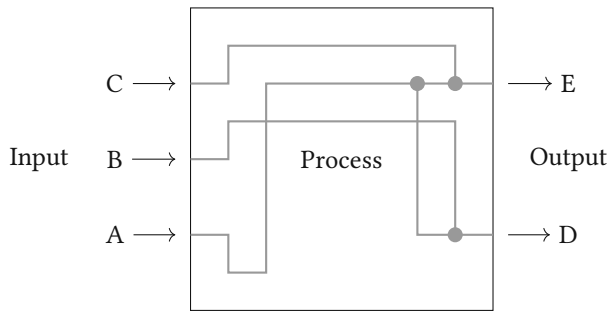


Figure 3.2: A process with 3 inputs (A, B, C) and 2 outputs (D, E). Information flows from the inputs to the process and propagates depending on its functionality. When information of an input is transferred to an output, there is an *information flow* between them.

If Lemma 1 is applied to a software system, this can be expressed as: the value of object A influences the value of object D, and this influence is visible by only observing the value of D. If D changes when A changes there is an information flow. This influence is possible via a generic process that transmits data from A to D.

[27] François Koeune and François-Xavier Standaert. "A Tutorial on Physical Security and Side-Channel Attacks". en. In: ed. by Alessandro Aldini, Roberto Gorrieri, and Fabio Martinelli. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005. (Visited on 03/11/2021).
[28] J. Ferrigno and M. Hlavac. "When AES blinks: Introducing optical side channel". In: *Information Security, IET* 2 (Oct. 2008).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

In a hardware system, specifically at gate level, information flow can be expressed as "The value of signal A influences the value of signal D, and this influence is observable only from signal D".

The presence of information, which is of interest for the verification engineer, at a specific point of the system or process is often associated with the term *taint* or *taint information*. When referring to Figure 3.2 we can illustrate the taint: If information is transferred from input C to the observed output D, then C is tainted. As it propagates through the process, parts of the process which have information of C as well, are tainted too. When the output D is being tainted by following this propagation of information from input C, we conclude that there is an information flow from the tainted input C to the output D. If output E is tainted too, there is also an information flow from C to E. Explained with Lemma 1, we can say that if we change input C, we may observe a change in outputs D and E. In the following, we just say "there is information flow from input C to output D", if the taint propagates from input C to output D.

The taint therefore acts as an indicator for the presence of information which is of interest to us. As information propagates through the system, so does the taint. Various methods exist to allow calculating this taint in a software or hardware system as will be discussed in Section 3.3.

Security domains

A security domain is formed by a set of objects that share the exact same level of security without a restriction in information exchange between them, as shown by Biba.[29] The level of a security domain is of interest when we focus on information flow between two or more domains and be expressed as a value. In a hardware design flow, the domains are assigned at the specification stage by the system architects.

Security domains can be put into a hierarchy by assigning them a security level. A formalized model incorporating security domains and levels is provided by Biba.[30] It introduces three security domains used for classifying information in documents:

- U = Unclassified,
- S = Secret,
- TS = Top Secret.

This example is known as "document classification" and describes the partitioning of documents. Some documents may be assigned with the domain *Unclassified* and can be read by anyone. Documents in the *Secret* domain can only be read by people who are authorized to access the domains *Secret* and *Top Secret*. Documents that are classified as *Top Secret* can only be accessed by people having access to the *Top Secret* domain. In the other direction, people having access to the *Secret* domain can access documents in both *Secret* and *Unclassified* domain. People with access to *Top Secret* can access in all three domains. The level of a security domain can be modeled as a function $sl(\text{DOMAIN})$ that returns the level as an integer for the given domain name. A hierarchy is then created by ordering security domains by their security level. Therefore, the actual value of the security level is of no interest, but the relative level, when dealing with multiple domains and levels, is relevant. We can compare the security levels of the three domains as shown in Equation (3.1).

[29] Biba, *Integrity Considerations for Secure Computer Systems.*
[30] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$sl(\mathrm{U}) < sl(\mathrm{S}) < sl(\mathrm{TS}) \tag{3.1}$$

We can visualize this hierarchy as shown in Figure 3.3.

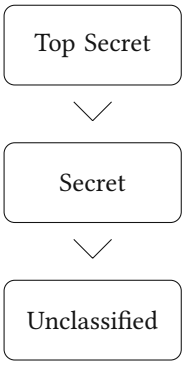


Figure 3.3: Domain hierarchy for the document classification example. Domains are ordered with the highest level on top. Information can only flow upwards in this hierarchy but never downwards to ensure no one in a domain with a lower level can read information from a higher level.

Security models

The purpose of a security model is to model the security objectives of a system. Some of them presented in Section 3.2. Using security policies, as we elaborate in Section 3.2, the model can be used to verify the security of a design. A security model also allows the designer to focus on security goals during the implementation stage of a system.

The most basic model is the so-called *access list.* In this model, every element of the system is assigned an access list that describes which other element of the system is allowed to read from or write to. In a basic access list there are no security domains and therefore no hierarchy. Access lists can be extended by creating groupings of elements. Access to information can then be granted to a whole group instead of single objects. This type of security model is found in modern operating systems and called *access control list (ACL)* in Unix based operating systems as well as Windows. File systems such as *New Technology File System (NTFS)* also use an ACL to control access on information.

We identify two models that partition elements of a design in security domains. These domains are ordered in a hierarchy. First, there is the Bell-LaPadula model described by Bell, D Elliott and LaPadula, Leonard J.[31] It defines three rules that manage information access between different levels of security domains. Given two security domains A and B with levels $sl(\mathrm{A}) > sl(\mathrm{B})$:

- **No-Read-Up** (also called *Simple Security Property*).
  An object in domain B, the lower domain, shall not read from any object in A, which has the higher level. This applies to all objects within domains with a level lower than an object's own security level. Referring to the "document classification" example shown in Section 3.2 we can express it as: "Information from TS shall not be read from S and U.

- **No-Write-Down** (also called $\star$ *(Star) Property*)
  An object in domain A shall not write to an object in domain B, or any other domain with a level lower than its own. The term $\star$ was originally used as a placeholder till they found a better name but after heavy use of the rule

[31] Bell, D Elliott and LaPadula, Leonard J, *Secure computer systems: Mathematical foundations.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

it was decided to keep its name.[32] Again, referring to the "document classification" example we can express the problem as "Information from TS shall not be written to S and U".

- **Access-Matrix** (also called *discretionary security property*). It is used to allow information flow between security domains as needed for the design's functionality. This property is allowed to break the other rules. It basically implements the function a multidimensional access list. In the "document classification" example this property could explicitly allow S to read from TS.

We conclude that the Bell-LaPadula model focuses on the confidentiality of information as information can never be transferred to lower level domains. A second model, described by Biba,[33] is handled as the reverse of the Bell-LaPadula model. It focuses on the integrity of information (data). Two rules must be checked before each information access:

- **No-Read-Down** prevents object A from reading from object B. It has no influence on object A writing to object B.

- **No-Write-Up** prevents object B from writing to object A. B can still read from A.

Figure 3.4 illustrates these models. It uses the three domains defined in Section 3.2: Unclassified (U), Secret (S), and Top Secret (TS). Solid arrows represent an information flow from a security domain to one with a higher security level, whereas dashed arrows show information flow from domains with high security level to domains with lower security level. Information flows that are prohibited by the *simple security property* and $\star$ *property* (Bell-LaPadula) are drawn using a dashed line. The flows prohibited by the model of Biba[34] are drawn using a continuous line. When featuring both models in order to ensure both integrity and confidentiality of all data no information flow would be possible between security domains as all flows would be forbidden.

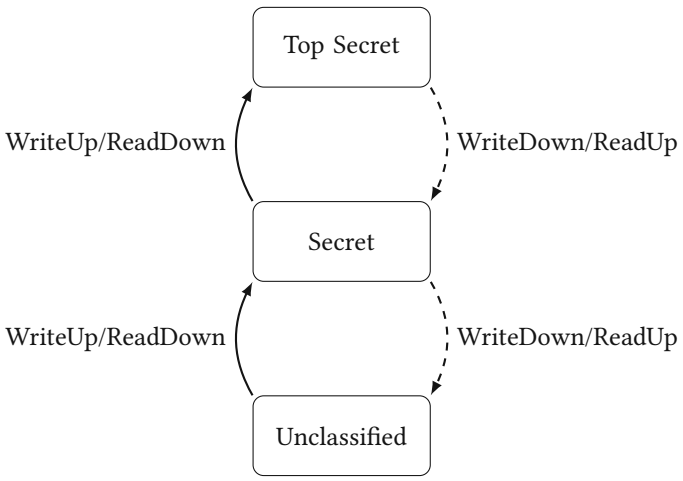


Figure 3.4: Combining the security models of Bell-LaPadula and Biba with the domain example from Section 3.2.

[32] Elliott D. Bell. "Looking Back at the Bell-La Padula Model". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005.
[33] Biba, *Integrity Considerations for Secure Computer Systems.*
[34] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Security policies

Security policies are used to specify security requirements of a system.[35] These policies will later be used to verify the DUT. We are able to deduce our properties by fitting the design into a security model as seen in Section 3.2. Information flow policies are derived from these security policies and provide a set of allowed and forbidden information flows. In this work, we use the security domains from Section 3.2 and models from Section 3.2 in order to formulate security policies, which are enforceable by information flow tracking techniques introduced in Section 3.3.

3.3 Information flow tracking

The idea of *information flow tracking (IFT)*, also called information *flow analysis* or *taint analysis*, is to follow the influence of information throughout a logic system. Available methods can be differentiated based on their implementation in either hardware or software and by their nature being static or dynamic. While dynamic IFT allows to analyze a process at runtime, static variants analyze the design before execution. Depending on the process to be verified an appropriate variant can be chosen.

Static and dynamic IFT

Using IFT we are able to verify a design for both timing and cache side channels. Further, data remanence side-channels can be identified using IFT methods. Implementations of IFT exist both for software and hardware. We can further differentiate between *static* and *dynamic* IFT. *Static information flow tracking (SIFT)* describes methods for verifying the design in the design process. After verification IFT logic that was added is removed. Violations of information flow policies are discovered by using verification tools that evaluate a design and information flows through the design. A modern example is *SecVerilog* by Zhang et al.[36] Security policies are introduced by extending the Verilog[37] hardware description language. It allows modelling information flows based on the design by adding additional logic equivalent to the original design but with a more complex logic function. The so created design is synthesized and verified using normal synthesis and verification tools. The verification goal is to verify that the values of the added logic satisfy the IFT policies of the system. After a successful verification process, additional IFT logic can be removed and implementation can be continued without any restriction. The finished design is therefore not altered by applying IFT. By adding logic that continuously monitors taint information to the system we can perform so-called *dynamic information flow tracking (DIFT)*. It observes the taint information at runtime to immediately identify information flows not matching the security policies. Different wordings exist for this taint information, such as *taint tag*, *taint label*, *security label* or simply *label*. DIFT is mostly used in software variants of IFT, but there are also implementations in hardware.[38] It adds constant overhead to each operation on data, which decreases overall system performance. Any information flow outside the specification is immediately detected and can be reported. The work of Zdancewic[39] sums up the features of static and dynamic IFT in more detail.

[35] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. ISSN: 1540-7993. Apr. 1982.
[36] Zhang et al., "A hardware design language for timing-sensitive information-flow security".
[37] .
[38] Tiwari et al., "Complete Information Flow Tracking from the Gates Up".
[39] Steve Zdancewic. "Challenges for information-flow security". In: Citeseer, 2004.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

IFT in software

Early research in computer security primarily considered access lists to maintain availability and confidentiality.[40] Myers and Liskov[41] present a decentralized model for information flow control that was adopted by ASBESTOS[42] and HiStar[43] by providing a modified Linux kernel. *Decentralized information flow control (DIFC)* consists of a trusted flow controller that is able to control the flow of information by untrusted software. Flume implements DIFC in user space and can be run on Linux or OpenBSD.[44] Using DIFC, security violations can only be caused by the trusted controller which can be verified in any possible way. Here, only the controller is verified, and it is assumed that it has control over any possible information flow. The rest of the designs is not verified for information flow due to this assumption. Eventually, design errors in the not verifiable part of the design could lead to information leaks even if the controller works properly.

To track information at design phase, JFlow[45] and its successor Jif[46] extend the Java programming language and allow to statically track information flow. After checking security properties, the Java engine executes the program without interfering with the added annotations. Without going deep into detail, Listing 3.1 shows a small part of a *Java information flow (Jif)* program as shown by Pullicino. Principals are entities in the program, access to data is granted by the authority construct. Arrows imply the association between entities. In this example, the entity *Owner* and all others can read data from the objects *cardNumber* and *number*, while only *Owner* is allowed to write to them.

```
1 public class Booking[principal Owner,
2       principal Operator] authority(Owner) {
3       private final String{Owner->*} cardNumber;
4       public Booking(String{Owner->*} number) {
5       this.cardNumber = number;
6       ...
7 }
```

Listing 3.1: Short example of a Jif class definition as provided by Kyle Pullicino. "Jif: Language-based Information-flow Security in Java". In: *CoRR* abs/1412.8639 (2014). Read access to the objects cardNumber and number are granted for all principals but only Owner can write to it.

This creates little code and execution time overhead. *Security enhanced Linux (SELinux)* is an extension of the Linux kernel and implements information flow tracking via its MLS (Multi-Level Security) feature.[47]

IFT in hardware

Similar to software, the principles of IFT can also be applied to a hardware design. This is generally done by analyzing the logical function of a circuit. In a logical function, information is transferred from the inputs to an output depending on the function. For a function with output Y and inputs A and B, information flows from A to Y and from B to Y, but not from A to B.

[40] Andrew C. Myers and Barbara Liskov. "A decentralized model for information flow control". In: *ACM SIGOPS Operating Systems Review* 31.5 (Dec. 1997).
[41] Ibid.
[42] Steve Vandebogart et al. "Labels and event processes in the Asbestos operating system". In: *ACM Transactions on Computer Systems* 25.4 (Dec. 2007).
[43] Nickolai Zeldovich et al. "Making information flow explicit in HiStar". In: *Communications of the ACM* 54.11 (Nov. 2011).
[44] Maxwell Krohn et al. "Information flow control for standard OS abstractions". In: *ACM SIGOPS Operating Systems Review* 41.6 (Oct. 2007).
[45] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*. 1999.
[46] CS Cornell. *Jif - Java Information Flow Tracking*. 2016.
[47] Inc. Gentoo Foundation. *SELinux/Information flow control*. [Online; state: March 4 2021]. 2014.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

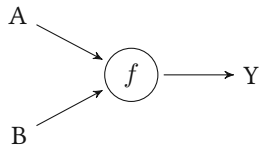


Figure 3.5: Information flow in a logical function f with inputs A, B and output Y.

The flow of information in a simple logic function f from its inputs to the outputs as shown in Figure 3.5 can be expressed as: "Y read from A and B", or "A and B write to Y" which are equivalent in context of a simple logic function. Hardware designs implement their logical function and therefore their functionality using logic gates that implement basic logic functions. Therefore, we are able to verify properties of a hardware design by analyzing its logic gates, subsequently reconstructing its logic function. We can deduce the gate level netlist from the design using available synthesis tools such as Yosys.[48] Using static analysis we are able to apply IFT to the so created logical function.

GLIFT

GLIFT is an implementation of IFT that operates on a hardware design represented as a gate level netlist. It allows to monitor every single bit inside the design. Each bit is assigned a *taint value* which represents the *taint* of this bit. Different wordings exist and are used in literature as shown before. Every gate in the hardware design represents a logical function whose IFT variant is previously known. The design is instrumented with additional gates that allow to track the flow of information through the original gate, depending on the gate type and the actual input values. This additional logic is generally named *shadow logic*.[49] Additional inputs control this shadow logic and called *taint inputs*. Their value indicates the presence of information. By setting these inputs we can choose information of which input flow shall be tracked starting from the gate's corresponding original inputs.

Example

Take a simple AND gate with two inputs A and B and one output O. By applying "Information flows from signal X to signal Y if and only if a change in the value of X can influence the value of Y"[50] to the case of a logic gate we can formulate:

Lemma 2 *Information is transferred from the input to the output of a boolean gate or logic cell if the change of the input alters the output value for at least one case.*

To mark the presence of information that shall be tracked, we instrument the design with additional logic, which processes the additional *taint inputs*. The actual value of these inputs control the taint information of the corresponding original inputs. Hu et al.[51] describe taint values by: "A logic variable is said to be tainted when its taint is logic true" Applying this to the AND gate, we add taint inputs A_t and B_t for each original input A and B, and an output O_t for the AND gate's original output O. The taint values are therefore tightly coupled to the original values in the design. One of the most basic works are provided by Tiwari et al.[52] and Hu et al.[53] They introduce methods to generate so-called *shadow logic* based on the netlist of the design under verification. Most of the implementation of GLIFT done in this

[48] Wolf, *Yosys Open SYnthesis Suite*.
[49] Hu, Armaiti, and Kastner, "Hardware Information Flow Tracking".
[50] Armaiti Ardeshiricham et al. "Register transfer level information flow tracking for provably secure hardware design". In: Mar. 2017, p.2.
[51] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking".
[52] Tiwari et al., "Complete Information Flow Tracking from the Gates Up".
[53] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Port	Value	Interpretation
A_t, B_t	0	No information at input A, B
A_t, B_t	1	Information at input A, B
O_t	0	No information is transferred to O
O_t	1	Information is transferred to O

Table 3.1: Interpretation of taint values for an AND gate.

thesis is based on their works. They further introduce a shadow logic function that represents the information flow at an AND gate as shown in Equation (3.2). This newly created shadow logic function can be illustrated as shown in Figure 3.6.[54] This additional created logic represents the flow of information at the *AND* gate. The function $sh()$ models the shadow function to the given logic function.

$$sh(f_{AND}) = AB_t \vee BA_t \vee A_tB_t \tag{3.2}$$

A	B	A_t	B_t	O_t
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Table 3.2: Complete truth table for the shadow logic function of an AND gate. Minterms are highlighted.

Creating shadow logic

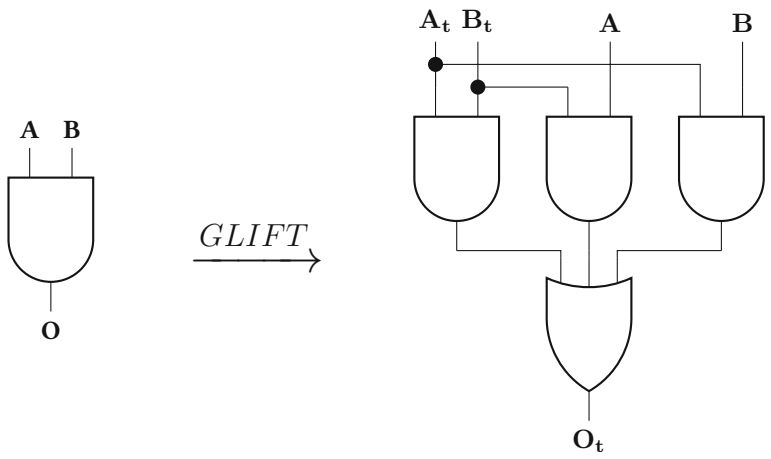
There are different ways to add GLIFT to a hardware design. One way to do so is the so called *constructive method*.[55] The implementation of a new, unknown cell is broken down to basic cells for which we already know its corresponding shadow logic. Tiwari et al.[56] provide IFT definitions for two input AND gates. Another method is the *brute force method*:

> "It works by changing the inputs to a logical function and observing what combinations can cause a difference in the output. For all combinations that cause a change at the output, a minterm is added to the shadow logic function."[57]

We chose the constructive method over the brute force method, since it is easy to implement in state-of-the-art synthesis flows, and the authors describe the constructive method as a "less computationally complex alternative to this brute force

[54] Ibid.
[55] Ibid.
[56] Tiwari et al., “Complete Information Flow Tracking from the Gates Up”.
[57] Hu et al., “Theoretical Fundamentals of Gate Level Information Flow Tracking”, p.6.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a) AND gate

(b) Shadow logic that tracks the taint information for an AND gate.

Figure 3.6: Shadow logic is created based on the original netlist. The taint output O_t implies the presence of information at the output O of the original gate. The corresponding truth table can be seen in Table 3.2. From W. Hu et al. "Theoretical Fundamentals of Gate Level Information Flow Tracking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011).

method". It can be summarized that the constructive method is more easy to implement but has some drawbacks that will be discussed further. The brute force method on the other hand is more resource consuming but is able to improve the precision of the created shadow logic function.

Precision

The constructive method is found to create more minterms than the brute force method due to assumptions made implicitly while creating shadow logic for gates that reassemble the given logic function. This leads to signals getting tainted in cases where it is not necessary. We call this behavior *imprecise* or *conservative* as described by Hu et al.[58] Section 3.3 shows this behavior while implementing shadow logic for the MUX cell. The most inexpensive, in terms of execution time and additional gates, approach to create shadow logic (that also creates the most conservative shadow logic) is given in Equation (3.3).

$$sh(f) = A_t \vee B_t \tag{3.3}$$

The taint value of an output of any gate is therefore 1 if any of the taint values of its inputs is 1. The actual values of the signals are not taken into account. Considering all the different approaches to implement IFT for our experiments we chose to develop a constructive GLIFT algorithm in the most precise way as possible. To further improve precision, we consider some findings from the brute force method on the MUX cell. Our implementation is therefore as precise as possible for a restricted set of cell types, as will be discussed in Chapter 5.

AND gates

The shadow logic function for the AND gate is provided by Tiwari et al. We take this function as given and use their findings to create shadow logic for all basic gate types that our software supports.

$$sh(f_{AND}) = AB_t \vee BA_t \vee A_tB_t \tag{3.4}$$

[58] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking", p.7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

NOT gates

We also know that NOT gates (also called INV or inverter) are transparent for information flow[59] , since an input change always results in an output change. Information is therefore always transferred from the input to the output. Changing the value of the input at an inverter gate does not influence the taint value as shown in Equation (3.5).

$$sh(f_{NOT}) = Y_t = A_t \tag{3.5}$$

Lemma 3 *The NOT gate is transparent for information flow.*

OR gates

We apply the findings and formulas from AND and NOT gates and combine it both with De Morgan's Law

$$(A \vee B) = \overline{(\overline{A} \wedge \overline{B})}. \tag{3.6}$$

We can now calculate the shadow logic function as shown in Equation (3.7). Hu et al.[60] came to the same conclusion.

$$sh(f_{OR}) = sh(A \vee B) = sh(\overline{(\overline{A} \wedge \overline{B})}) = sh(A \wedge B) = sh(f_{AND}) \tag{3.7}$$

XOR gates

XOR gates have a less complex shadow logic function compared to other gates (excluding the NOT gate). Every change at the input results in a change of the output. That means that information is transferred as soon as one of the inputs is tainted. Therefore, the shadow logic of a XOR gate can be calculated according to Equation (3.8).

$$sh(f_{XOR}) = A_t \vee B_t \tag{3.8}$$

NAND, NOR, XNOR gates

The fact that the shadow logic function of a NOT gate (with input A and output Y) is transparent to the output and applying Equation (3.5) is used to conclude:

Lemma 4 *Negating a signal does not influence its taint value, the shadow logic function represented by a NAND gate is therefore equivalent to that of an AND gate. This also applies to NOR/OR and XNOR/XOR gates.*

$$sh(f_{NAND}) = sh(f_{AND}) = AB_t \vee BA_t \vee A_tB_t \tag{3.9}$$

$$sh(f_{NOR}) = sh(f_{OR}) = sh(f_{AND}) = AB_t \vee BA_t \vee A_tB_t \tag{3.10}$$

$$sh(f_{XNOR}) = sh(f_{XOR}) = A_t \vee B_t \tag{3.11}$$

[59] J. Oberg et al. "Theoretical analysis of gate level information flow tracking". In: *Design Automation Conference*. ISSN: 0738-100X. June 2010.
[60] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

MUX cells

As already outlined, there are two variants to gather a shadow logic function for a new logic function: *constructive* and *brute force* as described by Hu et al.[61] Using the constructive method is easy by running our GLIFT implementation on a design consisting of a single MUX gate and preprocessing it to an *and-inverter graph (AIG)*. By synthesizing the MUX type to an AIG we gather a design consisting solely of gate types that already have an IFT implementation. The logic function, expressed as *disjunctive normal form (DNF)* of a MUX gate is given shown in Equation (3.12).

$$f_{MUX} = (A \wedge \overline{S}) \vee (B \wedge S). \tag{3.12}$$

Using the rules for creating shadow logic provided by Hu et al.[62] we can transform this into a shadow logic function shown in Equation (3.13).

$$sh(f_{MUX})_{\text{constructive}} = \overline{S}B_t \vee SA_t \vee \overline{A}BS_t \vee A\overline{B}S_t \vee ABS_t \vee A_tS_t \vee B_tS_t \tag{3.13}$$

The brute force method[63] generates a more precise shadow logic as shown in Equation (3.14). This function is derived directly from the truth table of a tainted MUX gate and misses the extra term ABS_t which would lead to a more conservative taint propagation.

$$sh(f_{MUX})_{\text{brute-force}} = \overline{S}B_t \vee SA_t \vee \overline{A}BS_t \vee A\overline{B}S_t \vee A_tS_t \vee B_tS_t \tag{3.14}$$

At gate level, this functionality is built using these cells: 12 AND, 4 NOT and 6 OR gates. Further 19 wires are added that interconnect the shadow logic gates.

Method	AND gates	NOT gates	OR gates	Wires
Constructive	10	3	6	18
Brute force	8	3	5	15

Table 3.3: Number of gates needed to create the shadow logic of a MUX gate with constructive and brute force method.

Flipflops, Memory

Memories and flip-flops are transparent for shadow logic. That means they affect taint information in the same way as they do with the original signals. A memory or flipflop gate, independent of the actual type, can therefore be mirrored into the shadow logic[64] and connected to the taint signals.

$$sh(f_{memory}) = f_{memory} \tag{3.15}$$

Brute-force method

The brute force method does not operate on the gate type but on the logic function. The difference gets clear when the design consists of multiple connected gates that form a unique logical function. See Section 3.3 where shadow logic

[61] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking".
[62] Ibid.
[63] Ibid.
[64] Tiwari et al., "Complete Information Flow Tracking from the Gates Up".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

is created for a MUX cell with both constructive and brute force method. This imprecision leads to information flows getting tainted when there is no actual information flowing. These false positives originate in so-called *imprecise* shadow logic as discussed in Section 3.3.

Precision of shadow logic

The purpose of shadow logic can be rephrased to:

Lemma 5 *When information flows from input A to output O and the taint input A_t is true, O_t is also true.*

The logic function of a design can be represented as a truth table. Here, every flow of information is indicated by a 1 at the tainted output. *Precise shadow logic* generates a 1 exactly for every state where information is transferred from a tainted input to a tainted output. *Imprecise shadow logic* creates additional 1s for tainted outputs that do not represent an actual information flow. We call these wrongly detected information flows *false positives*. Logic that generates these false positives is overall called *imprecise*.

Hu et al.[65] analyze the nature and consequences of imprecise shadow logic. They show that there are still benefits in creating imprecise shadow logic. Impreciseness adds less overhead in the design size (in terms of area used for this implementation in an *application-specific integrated circuit (ASIC)*, which can also be used as a metric for the amount of logic gates). This also effects the shadow logic generation and the verification time.

The distinction between a false and a true positive needs to be done according to the specification or using a golden model of the design. Each false positive therefore leads to an additional overhead. In case too many false positives are detected it may happen that the true positive is well distinguished between them since determination is done by the verification engineer. We get a better understanding of how false positives occur when comparing the two Karnaugh maps from both imprecise Equation (3.3) and precise (Equation (3.4)) shadow logic of an AND gate as shown in Figure 3.7. Additional minterms produced by imprecise shadow logic are marked red. These minterms result in the taint output being 1 which indicates an information flow where no actual information flow exists. Shadow logic that has such behavior is called imprecise or conservative. Using imprecise shadow logic also has benefits. The created shadow logic is usually less complex due to the reduced amount of terms in the logic function. Complexity in IFT logic is measured in the amount of used gates of the logic function. The number of false positives is also used as a metric in IFT publications[66] when different IFT methods are compared.

False positives are also often introduced when using the brute force method to construct the shadow logic. When constructing shadow logic for a MUX cell we analyze the logic function of the MUX

$$f_{MUX} = (A \wedge \overline{S}) \vee (B \wedge S) \tag{3.16}$$

and reconstruct its functionality consisting of AND, INV and NOT gates as shown in Figure 3.8. This circuit is functionally equivalent to a MUX cell and is the default way to build a MUX in hardware (gates can still be mapped to other logic gates as needed). When we further analyze the functionality of the MUX we see that signals from the AND gates

[65] Wei Hu et al. "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking". In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD '16. Austin, Texas: Association for Computing Machinery, 2016.

[66] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking"; W. Hu et al. "Expanding Gate Level Information Flow Tracking for Multilevel Security". In: *IEEE Embedded Systems Letters* 5.2 (June 2013); Ardeshiricham et al., "Register transfer level information flow tracking for provably secure hardware design"; Tiwari et al., "Complete Information Flow Tracking from the Gates Up"; Armaiti Ardeshiricham. "Verification and Synthesis of Information Flow Secure Hardware Designs". PhD Thesis. UC San Diego, 2020.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

BB_t \ AA_t	00	01	11	10
00				
01		1	1	1
11		1	1	1
10		1	1	

(a) Karnaugh map for precise AND shadow logic $sh(f_{AND}) = AB_t \vee BA_t \vee A_tB_t$.

BB_t \ AA_t	00	01	11	10
00		1	1	
01	1	1	1	1
11	1	1	1	1
10		1	1	

(b) Karnaugh map for imprecise AND shadow logic $sh(f_{AND}) = A_t \vee B_t$.

Figure 3.7: Comparing precise and imprecise shadow logic for the AND gate. Additional minterms lead to information flows get falsly detected when there technically is no information flow. From: Wei Hu et al. “Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking”. In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD ’16. Austin, Texas: Association for Computing Machinery, 2016.

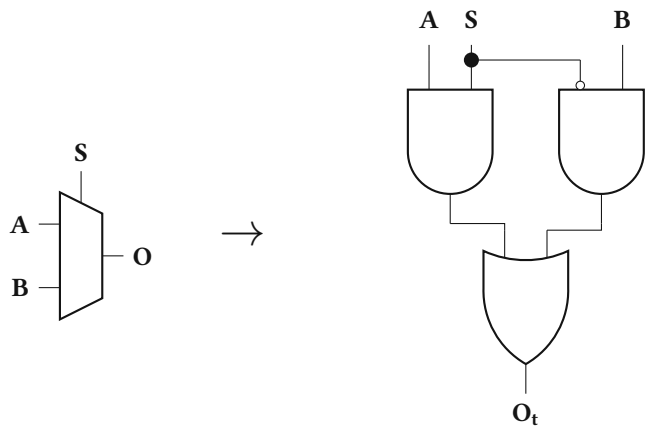
to the OR gate can never be true at the same time since they are switched by the S signal. The "11" state is therefore never reachable and can be treated as a "don’t care" state. With this knowledge we can exchange the OR gate with an XOR and still yield the same functionality. The shadow logic function

$$sh(f_{OR}) = (A \wedge B_t) \vee (B \wedge A_t) \vee (A_t \wedge B_t) \tag{3.17}$$

can therefore be replaced by

$$sh(f_{XOR}) = A_t \vee B_t \tag{3.18}$$

as shown in Figure 3.9. Using the less complex shadow logic function of a XOR gate to construct the shadow logic for a MUX cell results in a both less complex and more precise shadow logic function.

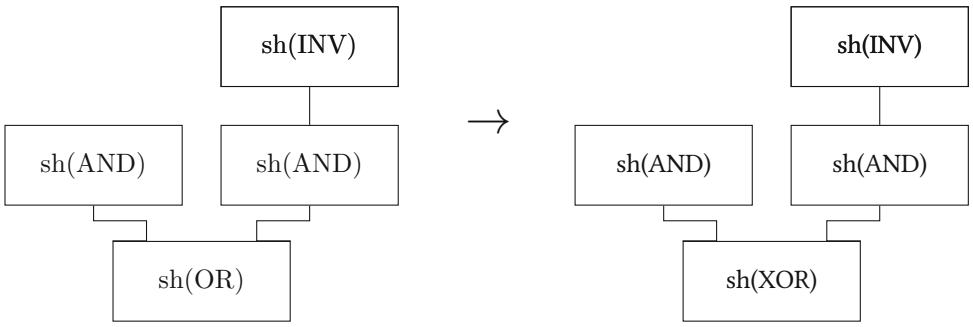


(a) Graphical representation of a "2 by 1 MUX".

(b) Logic of a MUX cell is constructed with AND, OR and INV gates.

Figure 3.8: Implementation of the logic function of a MUX gate.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN



(a) Shadow logic that reassembles the information flow at a MUX cell.

(b) Optimized shadow logic based on an XOR instead of an OR for the output gate.

Figure 3.9: The inputs of the OR never reach the state 11. Therefore, a XOR gate can be used whose shadow logic is simpler but more precise that of the OR gate as shown by Wei Hu et al. "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking". In: *Proceedings of the 35th International Conference on Computer-Aided Design*. ICCAD '16. Austin, Texas: Association for Computing Machinery, 2016.

RTLIFT

Other than GLIFT, *register transfer level information flow tracking (RTLIFT)* analyzes the design at the RTL level rather than the gate level netlist. In a design flow the RTL is created before the design is synthesized to the gate level. By applying IFT methods to the word level the verification can be done in an early design phased. The method described by Ardeshiricham et al.[67] claims to

> "show that RTLIFT achieves over 5× speedup in verification performance as compared to gate level IFT while minimizing the required effort for the designer to verify security properties on RTL designs."[68]

They also allow the verification engineer to choose between precise and conservative IFT shadow logic creation. Their method analyzes the design as an *abstract syntax tree (AST)* and introduces additional shadow logic. This design is then verified and eventually modified if verification fails. After a successful verification the additional logic is removed from the design. This is achieved by analyzing objects like a MUX and replacing it with an expression that also tracks the information flow as shown by Ardeshiricham et al.[69] RTLIFT showed to be more execution time friendly than GLIFT since it ditches the synthesis step to create the gate level netlist. Although RTLIFT offers verification at a higher level of abstraction the implementation for gate level cells was chosen to create comparable results. We can compare our method to others[70] which would not be doable due to missing published RTLIFT implementations. Furthermore, the chosen synthesis tool Yosys offers easy to use ways to manipulate the design at gate level which made it best way to implement a prototype pass.

3.4 PGSL

PGSL is primarily used to describe design (functional) graphs in a unique and implementation independent way. The language is described formally and provided as *Backus-Naur form (BNF)* and shown in Figure 3.10. It was published by Krieg.[71] PGSL's main goal is to describe patterns in a *cell adjacency graph (CGA)*. These patterns are then fed to a searching algorithm that evaluates the design in order to find occurrences of these patterns. Patterns can be created

[67] Ardeshiricham et al., "Register transfer level information flow tracking for provably secure hardware design".
[68] Ibid.
[69] Ibid.
[70] Hu et al., "Theoretical Fundamentals of Gate Level Information Flow Tracking".
[71] Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

for known malicious design parts. By expressing the design as a PGSL pattern abstraction is introduced which allows various implementations to be described using a single pattern. Structural variability therefore does not influence the pattern. The idea behind PGSL is that the verification engineer creates a set of patterns that describe known malicious design parts and then verifies the design for not containing any matching occurrence of those patterns. In a verification process many more steps are taken to try to prove the security of the design. To further decrease the verification time we chose to use this language, that the verification engineer already mastered, to describe IFT policies. Other IFT variants enforce a new syntax that describes information flow. SecVerilog[72] for example extends Verilog and adds new syntax in order to describe valid information flows.

PGSL is very versatile, we therefore do not need its full set of features, while the created patterns still represent a valid PGSL pattern. The process of creating and using these patterns in order to introduce a security domain hierarchy and formally verify a hardware design is discussed in Chapter 4 Methodology.

LFSR Example

An example used by the author demonstrate the use of PGSL by implementing a pattern for a *linear feedback shift register (LFSR)*. LFSRs are used in HT to leak confidential information.[73] They consist of multiple XOR gates connected in series. The amount of connected XOR gates can vary which makes it hard to detect such structures using a search algorithm in the design. The pattern shown in Listing 3.2 is able to detect a generic LFSR, built by a flip-flop connected to one or more XOR gates in series by using operators for inner and outer connections.

```
[[\$dff->(\$xor>+)|+]]
```

Listing 3.2: PGSL pattern used for detecting a LFSR.

Structure

Figure 3.10 shows the BNF of PGSL. Its structural elements are shown in Table 3.4.

[72] Zhang et al., "A hardware design language for timing-sensitive information-flow security".
[73] Krieg, Wolf, and Jantsch, "Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
⟨pattern⟩ ::= ⟨opt_p_label⟩ ⟨expression⟩ ';'

⟨expression⟩ ::= ⟨node_identifier⟩ ⟨opt_quantification⟩ ⟨opt_e_label⟩
   | '(' ⟨expression⟩ ')' ⟨opt_quantification⟩ ⟨opt_e_label⟩
   | '[[' ⟨expression⟩ ']]' ⟨opt_e_label⟩
   | ⟨expression⟩ '->' ⟨expression⟩
   | ⟨expression⟩ '||' ⟨expression⟩
   | ⟨e_label⟩

⟨quantifier⟩ ::= '#' | '~' | '+' | '*'

⟨inner_connector⟩ ::= '>' | '|'

⟨opt_quantification⟩ ::= ⟨empty⟩ | ⟨quantification⟩

⟨quantification⟩ ::= ⟨inner_connector⟩ ⟨quantifier⟩
   | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ ',' ⟨UNSIGNED⟩ '}'
   | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ '}'
   | ⟨inner_connector⟩ '{' ⟨UNSIGNED⟩ ',' '}'

⟨opt_e_label⟩ ::= ⟨empty⟩ | ⟨e_label⟩

⟨e_label⟩ ::= ':' ⟨STRING⟩

⟨opt_p_label⟩ ::= ⟨empty⟩ | ⟨p_label⟩

⟨p_label⟩ ::= ⟨STRING⟩ '='

⟨node_identifier⟩ ::= ( '$' | '\' ) ⟨STRING⟩

⟨STRING⟩ ::= [a-zA-Z_0-9]+

⟨UNSIGNED⟩ ::= [0-9]+
```

Figure 3.10: The grammar of *pattern graph specification language (PGSL)* in *Backus-Naur form (BNF)* Christian Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Feb. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Blocks	
(	Block begin
)	Block end
Outer operators	
->	Serial connection
||	Parallel connection
[[	Loop begin
]]	Loop end
Inner operators	
>	Serial
|	Parallel
Quantifiers	
{n}	Match exactly n times
{n,}	Match at least n times
{n,m}	Match from n to m times
#	Match even number of times
~	Match odd number of times
+	Match one or more times
*	Match zero or more times
?	Match zero or one time
Operator Precendence	
Inner > parallel outer > serial outer (+ loop)	

Table 3.4: Structural language elements of PGSL Christian Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Feb. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.5 Hardware verification

Hardware verification or simply *verification* is used to guarantee the behavior of a given hardware design. The design to be evaluated is generally called DUT or *device under verification (DUV)* when talking about verification. Classical verification and validation methods include the creation of a "testbench" module that stimulates the DUT based on the design specification. The stimulus needs to be created according to the design specification (which can be informal, oral, expressed with state machines, ...). The testbench itself is a not synthesizable design that instruments the clock and other signals of the DUT. It is also connected to outputs of the design and test them for being the expected values using assertions. A simulator then evaluates the design and outputs data in a user-friendly way. This approach is limited in the number of clock cycles that can be evaluated. Depending on the number of input values and internal states the needed computation time can quickly exceed what is acceptable or even doable.

Consider a system having two 128-bit inputs, for example an AES128 encryption core that has both 128-bit key and 128-bit data as inputs. Simulating on a system that takes $1ps$ per input value we would spend $10^{-12}s * 2^{128} \approx 3.4 * 10^{26}s$, which is approximately $1.1 * 10^{19}$ years, for doing an exhaustive simulation for every possible plaintext input. And that is without changing the key or having internal states that would increase this time to even bigger times. Typical hardware trojans are triggered by rare events, for example a specific plaintext input. In order to detect these type of trojans an exhaustive simulation would be needed, otherwise the chances of detection are very low. A more modern way of verifying the design against its specification is to use formal verification.

Formal verification

Traditional testing and simulating, as shown before, is almost never feasible due to the amount of input and state combinations. This is where formal verification steps in.

> "Formal verification, in contrast to testing, uses rigorous mathematical reasoning to show that a design meets all or parts of its specification. This requires the existence of formal descriptions for both the specification and implementation".[74]

Gupta[75] defines formal verification as "formally establishing that an implementation satisfies a specification". The specification needs to be formulated in a way, so that verification can be done based on it. Nowadays, a typical way to do so is to formulate the specification in a *hardware description language (HDL)*. SystemVerilog[76][77] is an extension to the widely used Verilog HDL. The so formulated properties of the design can then automatically be verified by connecting them to the original design. Various tools exist that ease the use of formal verification such as SymbiYosys.[78] It uses a design instrumented with a specification and verifies it using methods such as SAT solving and bounded model checks (BMC). Using a configuration script we are able to fully control what steps are done to verify the design.

In this thesis we use formal verification to prove that the tainted output of interest, depending on our policy and therefore the domain of interest, is always equal to "0", implying that no information is transferred to the corresponding output.

[74] Christoph Kern and Mark R. Greenstreet. "Formal verification in hardware design". In: *ACM Transactions on Design Automation of Electronic Systems* 4.2 (Apr. 1999).
[75] Aarti Gupta. "Formal hardware verification methods: A survey". In: *Formal Methods in System Design* 1.2-3 (Oct. 1992).
[76] Mehta, *SystemVerilog Assertions and Functional Coverage.*
[77] Vaibbhav Taraate. *SystemVerilog for Hardware Description: RTL Design and Verification.* Springer Nature, 2020.
[78] Wolf, *SymbiYosys.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$\forall O_t \in \text{OutputDomain} : O_t = \text{"0"} \tag{3.19}$$

Using formal verification we will be able to prove that no information flow occurs within n clock cycles, with n being a user adjustable number.

Yosys

Wolf[79] is an open source synthesis tool developed to synthesize hardware designs in high level HDL Verilog. Verilog2005 is natively supported as well as some basic SystemVerilog constructs. There is also a commercial version of Yosys sold by SymbioticEDA. [80] This commercial version features more complex synthesis tools and also a *very high speed integrated circuit hardware description language (VHDL)* parser. It is implemented in C++ and can be extended by self written plugins. Typical operations done on a hardware design are provided using so-called "passes". The design itself is internally represented in *register transfer level intermediate language (RTLIL)*, a set of C++ objects that reassemble the design in memory. There exists a Python wrapper called Pyosys[81] that enables the access of internal data structures using Python. We provide more details on the internals of Yosys, its usage and the usage of Pyosys are provided in Chapter 5 Implementation.

Symbiyosys

SymbiYosys[82] is an open source hardware verification frontend operating on the open source synthesis tool Yosys. It allows to load a design and verify it using different tools and methods. The workload for the verification engineer is kept low by supporting various methods out of the box. The complete configuration is done by creating a short config file in their introduced *.sby* format. Their website [83] shows many example configuration and runs.

3.6 Related work

SecVerilog

SecVerilog, developed by Zhang et al.,[84] performs GLIFT on Verilog hardware designs. They describe it as:

> "SecVerilog is Verilog, extended with expressive type annotations that enable precise reasoning about information flow".[85]

Their method modifies the Verilog HDL by adding statements on security levels. An additional script language is used to describe transitions between those levels and forms their information flow policy. These policies basically describe the allowed information flow. Their main focus is on timing based side-channels and cache attacks. They formulate security policies using a description language used only for this application. Here we identify the possibility of using an already existing description language so the verification engineer.

[79] Wolf, *Yosys Open SYnthesis Suite.*
[80] https://www.symbioticeda.com
[81] Tutzer et al., "Python Wraps Yosys for Rapid Open-Source EDA Application Development".
[82] Wolf, *SymbiYosys.*
[83] https://symbiyosys.readthedocs.io
[84] Zhang et al., "A hardware design language for timing-sensitive information-flow security".
[85] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Example

In SecVerilog security levels are represented in a lattice. It uses a default lattice of 4 levels: HIGH, D1, D2 and LOW as shown in Figure 3.11. Their method allows information to flow to higher security levels than itself. Flows to lower

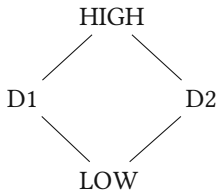


Figure 3.11: Default security lattice used by SecVerilog. Elements are assigned in security domains LOW, D1, D2 and HIGH. Information can only flow up, therefore information can flow from LOW to HIGH, D1 to HIGH and D2 to HIGH. Danfeng Zhang et al. "A hardware design language for timing-sensitive information-flow security". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015)

levels are prohibited. A flow from level D1 to D2 is therefore forbidden since the only information from D1 can only flow to HIGH but not from HIGH to D2.

They describe it as:

> "A lattice of security levels restricts information flow: information with lower security levels can flow to information with higher security levels, but the opposite direction is prohibited."[86]

To add information about security levels the design has to be enhanced with information about those levels. Each element in the design has a *label* that defines its security level. These labels have to be defined in both the source files and in an additional definition file where functions can be defined that describe how labels change at different elements. For example the label at the output of an MUX gate changes with the select input. Both domain definitions and such functions for switching domains are expressed in a separate file. This definition is expressed using statements of the SMT-LIB language. After adding information about security levels and transitions between them, their tool is used to compile the source files. Output files can than be verified using the Z3 solver.[87]

To perform GLIFT they modify the original design and add information about security domains to the design itself as shown in Listing 3.3. This module definition is provided by the authors of SecVerilog.[88] These additional information is automatically removed by their tool after a successful verification. We see the domain definition in braces for every element in the design. The definition of the domain for the register *out* is different. The expression `{Domain sel}` defines a function `Domain` that decides on the security label `sel`. The function itself is declared in an external file as shown in Listing 3.4.

After adding domain information to the design an additional file is created that introduces the security labels and the information flow policy as shown in Listing 3.5. This policy reassembles the lattice shown in Figure 3.11. For verifying a MUX design the verification engineer has to modify five lines in the designs source files. Additionally, two external files are created containing nine lines of code for defining the security lattice and the functions for domain changes. This amount increases with de complexity of the lattice and the amount of elements in the design as well as label changes that are specified with functions.

[86] Ibid.
[87] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
[88] Zhang et al., "A hardware design language for timing-sensitive information-flow security".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
module Two_Input_Mux
(
  input {D1}          in1,
  input {D2}          in2,
  input {L}           sel,
  output {Domain sel} out
);

reg {Domain sel} out;

always @ (*) begin
  if (sel == 1'b0)
    out = in1;
  else
    out = in2;
end
endmodule
```

Listing 3.3: MUX example provided by the authors of SecVerilog. Danfeng Zhang et al. "A hardware design language for timing-sensitive information-flow security". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015)

```
; Domain(0)=D1, Domain(1)=D2
(declare-fun Domain (Int) Label)
(assert (= (Domain 0) D1))
(assert (= (Domain 1) D2))
```

Listing 3.4: Function definition for a security label change in SecVerilog.

```
; lattice elements
( declare-fun LOW () Label )
( declare-fun HIGH () Label )
( declare-fun D1 () Label )
( declare-fun D2 () Label )

; lattice structure
( assert ( forall (( x label )) ( leq LOW x )))
( assert ( forall (( x label )) ( leq x HIGH )))
```

Listing 3.5: Security policy definition used in SecVerilog.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 4

Methodology

As shown in Chapter 3, *information flow tracking (IFT)* can be used to detect information flows in a hardware design by examing the gate level netlist of the design.

The design and its specification can be utilized to segment the design into multiple security domains. Inside a security domain, there is no restriction on information flow. Information flow between two domains is either allowed or not, depending on the security policies. Such a policy is built upon the specification of the design itself. Security models such as Bell-LaPadula[1] or Biba[2] help to create policies for a design specification that tests for confidentiality or integrity of data.

We identified the lack of a universal method for describing security domains and formalizing security policies. Methods like *SecVerilog*[3] introduce security policies by adding additional syntax that describes allowed information flow. These statements are injected directly into the Verilog source code and they define an additional script language that allows to model more complex information flow, like at a MUX. The verification engineer has to take extra effort to model the information flows depending on the design's specifications. This effort has to be taken additionally to the design process itself. At the end of this chapter we compare our method to SecVerilog by applying it to our example design.

During the creation of this work we could not identify a widely used description language that is capable of modelling security domains and information flows. However, there are existing languages which are used in other fields of hardware verification such as *pattern graph specification language (PGSL)*.[4] This versatile language can be extended and allows usage outside its original intended use case.

We propose a method that is used to easily define security domains and model the flow of information between them. This is done by creating a pattern that is fully compatible to PGSL as presented in Section 3.4. Our method then evaluates this PGSL pattern in order to create a *test case* that is then used to verify the design against this specification. This is done by using *gate level information flow tracking (GLIFT)*. Information on the security domains is introduced by using available Verilog syntax. Our method does this without altering the original hardware design in any way. As shown in Chapter 2 Motivating example this is possible by creating a wrapper module that connects to the original design. In this

1 Bell, D Elliott and LaPadula, Leonard J, *Secure computer systems: Mathematical foundations.*
2 Biba, *Integrity Considerations for Secure Computer Systems.*
3 Zhang et al., "A hardware design language for timing-sensitive information-flow security".
4 Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

chapter, the abstract document classification example presented in Section 3.2 Security domains, is used to demonstrate the working principle our method. Section 4.5 PGSLIFT example introduces an example design and shows every aspect of the flow and its usage.

PGSL is a language created to define abstract graphs in order to find them in a hardware design. It is proposed to be used in a hardware verification flow in order to detect potential malicious hardware parts. Its definition is shown in Figure 3.10 and its structural elements in Table 3.4. We adapt PGSL to define a domain hierarchy similar to that what is shown in Figure 3.3 by using a subset of it. The hierarchy itself needs to be created based on the design's specification. It is formalized using PGSL syntax. This domain hierarchy represents a security policy against which we verify the design. For this process we use GLIFT and formal verification. We define an abstract security policy via PGSL pattern as presented in Listing 4.1. It forms a valid PGSL pattern as described by Krieg.[5] The used elements of PGSL only form a small subset of the available syntax. We make use of three elements, as shown in Table 4.1.

Element	Original usage	Our usage
`p_label =`	Pattern name	Policy name
`node_identifier`	Generic node name	Security domain name
`->`	Outer connection	Domain level hierarchy indicator

Table 4.1: PGSL elements used to describe a domain hierarchy.

```
<p_label> = <node_identifier> -> <node_identifier> -> ... ;
```

Listing 4.1: A PGSL pattern is used to describe a domain hierarchy. Two or more *node_identifier* elements describe the security domains.

Using this pattern style enables us to create a pattern to verify the design as presented in Chapter 4. In reference to the document classification example as described by Biba[6] and shown in Section 3.2 Security domains, we extract three security domains U, S and TS. The hierarchy of these domains is graphically presented in Figure 3.3. We create the pattern shown in Listing 4.2 by applying the names of the security domains. This pattern is used to verify confidentiality by introducing a hierarchy of the involved security domains. Using this pattern enables us to verify that information is neither transferred from TS to S or U nor from S to U. The additional character "\\" is predefined by PGSL grammar for objects of type *<node_identifier>* since every element in Yosys' RTLIL is preceded by "\\" or "$" as described in Yosys manual.[7]

```
Policy0 = TS -> S -> U ;
```

Listing 4.2: *PGSLIFT* pattern that describes a policy for tracking information flow for the document classification example.

4.1 PGSL pattern as security policy

In order to verify the design, we need to add information of security domains to the design. We do this by using the well known Verilog *attributes*. Sutherland[8] describes them as:

[5] Krieg, "Pattern-Based Hardware Trojan Characterization for Design Security Assessment".
[6] Biba, *Integrity Considerations for Secure Computer Systems*.
[7] Wolf, *Yosys Open SYnthesis Suite*, p. 30.
[8] Sutherland, *Verilog® HDL*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

> "An attribute specifies special properties of a Verilog object or statement, for use by specific software tools, such as synthesis. Attributes were added in Verilog-2001. An attribute can appear as a prefix to a declaration, module items, statements, or port connections. An attribute can appear as a suffix to an operator or a call to a function. An attribute may be assigned a value. If no value is specified, the default value is 1".[9]

These attributes consist of a string that is the attributes name and a value which can also be a represented as a string. An attribute begins with `(*` and ends with `*)`. We decided to name attributes, that assign security domains to an input or output, `sec_domain`. Listing 4.3 shows how an input In0 is assigned to the security domain Secure. There exist no

```
(* sec_domain = "Secure" *) input In0;
```

Listing 4.3: The input In0 is marked with the security domain "Secure".

predefined attributes in the Verilog-2001 standard. The single use of them is adding information that software tools involved in the process can evaluate and manipulate. In order to introduce these attributes the original design's port list could be altered. Another way that does not alter the source code is to introduce a *wrapper* module as shown in Figure 4.1. It connects to all ports of the *design under test (DUT)* and adds Verilog attributes to the connective wires.

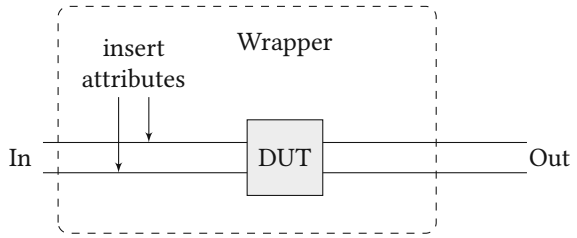


Figure 4.1: A wrapper module connects to all inputs and outputs of the original design. The original source code is not altered by this method.

This wrapper module is created using standard Verilog syntax and does not require the addition of code in the original source files. Unlike other works, for example *SecVerilog*,[10] there is no need of removing anything from the source code after the verification process finished. In addition, we are able to split signals into single bits by using the wrapper in order to apply security domains. This granularity, which can be used if needed, is also one of the benefits when using GLIFT.

Information flow tracking (IFT) tracks information by adding shadow logic that reassembles the original design. Information is tracked by monitoring the outputs of shadow logic cells. They represent the presence of information at the original cell. When creating shadow logic, we also create additional input and output ports. While output ports imply if information flows to the original output port, taint input ports need to be instrumented. Shadow logic inputs are used to mark the presence at the design's input. In order to verify that no information flows from an input of security domain TS to an output in domain U, we therefore need to assign correct values to the taint inputs. From Table 3.1 we see that the value 1 at a taint input represents how information at the original input shall be tracked. Identifying an information flow from domain TS to U requires the need to instrument the taint input of all inputs in domain TS with 1. All other taint inputs get assigned 0. To verify that no information flows to domain U, we need to monitor all taint outputs in domain U and ensure they are 0 all time.

[9] Ibid.
[10] Zhang et al., "A hardware design language for timing-sensitive information-flow security".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Lemma 6 *We can reason that there is no information flow from an input A to output B if the taint input $A_t = 1$ and the taint output $B_t = 0$ for every possible input and internal state combination.*

We can express this property as an *assertion* which will be used in the verification process later on. This assertion can be verbalized as:

Lemma 7 *We assert that all taint outputs in the domain of interest equal 0 which indicates no flow of information.*

Using formal verification methods, we are able to verify this property, as shown later.

Test cases

Information flows have an origin and a destination. Information is injected into the hardware design by an object and is possibly transferred to one or more objects. Consider a design featuring two inputs A, B and two outputs C, D. We can test it for an existing information flow from input A to output C if we instrument the taint input A_t with 1 and request the taint output C_t to stay 0. The so instrumented shadow logic is able to detect this information flow. Other taint inputs need to be set to 0 since we are only interested in the information flow originating in A while other taint outputs are of no interest whilst testing for this information flow.

We introduce a *test case* that represents the information flow to be tested, for example from point A to point B, the correlating taint input values, the asserted taint output values and the design itself. A test case T is affiliated to an information flow F which the design will be tested for.

$$F = (\text{Input domain} \rightarrow \text{Output domain}) \tag{4.1}$$

We can represent such a test case as a 4-tuple:

$$T = \langle F, I_t, O_t, D \rangle \tag{4.2}$$

where:

$$\begin{aligned} F &= \text{information flow to be tested,} \\ I_t &= \text{taint input values,} \\ O_t &= \text{taint output values,} \\ D &= \text{hardware design to be tested.} \end{aligned}$$

As shown before, we can extract the values of taint inputs by evaluating the PGSL pattern that describes the domain hierarchy and match it with the design. The taint inputs and outputs are therefore a function of the design, the information flow to be tested and the policy.

$$\langle I_t, O_t \rangle = f(F, P, D) \tag{4.3}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

We can use this to create a second presentation of the test case as an equivalent 3-tuple:

$$T = \langle F, P, D \rangle \tag{4.4}$$

where:

$F =$ information flow to be tested,

$P =$ security policy, as PGSL pattern, describing the domain hierarchy,

$D =$ hardware design with domain information

Depending on the number of security domains the number of possible information flows and therefore the number of theoretical test cases varies. By examining the pattern `Policy0 = TS -> S -> U` shown in Listing 4.2 we can extract three test cases: `TS->S`, `TS->U` and `S->U`.

The test case `TS->U` is not optional as it can not be substituted by `TS->S` and `S->U`. This assumption would only be suitable if the domains consisted of only one object. For example, there could be a flow from an object `TS_0` in `TS` to an object `S_0` in `S`. Furthermore, if there were two objects `S_0` and `S_1` in `S`, there could be a flow from object `S_1` ↪ to object `U_0` in domain `U`. We can reduce the number of test cases by determining which domains are inputs and outputs of the design. This results in eliminating the flows that can not occur since there is no matching input output combination.

Our main goal is identifying specific information flows from one input domain to one output domain. We create test cases that allow testing for this specific information flow. We neither have interest in flows from and to other security domains, nor are we interested in the original output value since the flow of information is indicated by the taint output. Using these findings, we can now remove those parts of the design that are not needed to detect the information flow.

Lemma 8 *We can isolate parts of the hardware design that have an influence on the taint output value of interest by removing all design parts not connected to it.*

The design used in the verification process can therefore be significantly smaller than the original design, in terms of logic cells. As discussed in Section 3.3, the execution time of both shadow logic creation and verification time are used as measures to compare IFT methods. The usage of a smaller design allows us to reduce execution times and minimize resource consumption. As shown in Chapter 5, we identified two methods to segment a hardware design. Non-taint outputs can simply be removed from the design. Synthesis tools such as Yosys offer methods to both remove these *ports* and also identify and remove current unconnected parts. By using these methods, we extract the *input cone* of the relevant taint outputs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.2 GLIFT

We verify the created security policy using GLIFT. As we presented in Section 3.3, this is done by creating additional shadow logic that tracks the information flow across the hardware design. It is based on the gate level netlist of the design. It uses so-called *taint* signals that represents the presence of information at every point in the design.

At gate level, the design is implemented with solely gate level cells. A design can either be designed at gate level or synthesized from a higher level of abstraction. Shadow logic is created using the constructive method as shown in Chapter 3 Preliminaries. This means that we add additional logic for every gate which allows to track the information flow by monitoring its output values.

We created a plugin for the synthesis tool Yosys that creates shadow logic for a given hardware design. This makes it more versatile and allows the usage outside our proposed method. Yosys implements its core synthesis functionality as so-called *passes*. The resulting tool is therefore named *GLIFT pass*. Details on the concrete implementation are provided in Chapter 5 Implementation. The hardware design, extended with shadow logic, can then be instrumented with taint input values based on the test case.

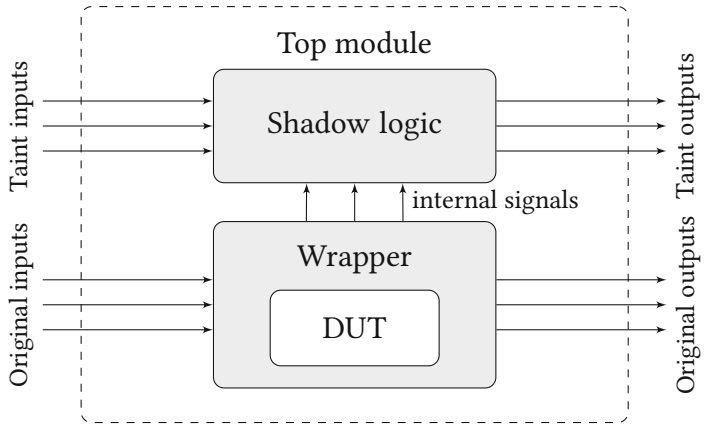


Figure 4.2: The wrapped DUT module is analyzed, and shadow logic is created. It adds additional inputs and outputs with it which we instrument based on the test case. Shadow logic also connects to the internal signals of the DUT to use logic with a high precision.

4.3 Formal verification

In order to prove the absence of an information flow, we need to prove that the taint output value is 0 at all times. We chose to use the open source formal verification tool SymbiYosys, which is based on the open source synthesis tool Yosys. To comply with the SystemVerilog standard, we introduce a separate Verilog module that solely holds assumptions, which sets the taint inputs. It further contains assertions in order to instrument the taint outputs for the formal verification process. We use a *SystemVerilog assertion (SVA)* construct to introduce these assumptions and assertions. We name this module "verification *intellectual property (IP)*" and store it using the suffix "_ sva" to imply its function. An example of this verification IP is shown in Listing 4.4.

This verification IP is connected to the DUT in a new top module, which is automatically generated by our script. An example of such a top module is shown in Listing 4.5. The connecting process is illustrated in Figure 4.3.

The verification process itself is controlled using a configuration file, an example of such a configuration file is shown

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
module simple_bus_sva(
  input i0,
  input i0_t,
  input i1_t,
  input i1,
  input o0_t,
  input sA_t,
  input sA,
  input sB_t,
  input sB
);
  always @(*) begin
    assume(i0_t == {1{1'b1}});
    assume(i1_t == {1{1'b0}});
    assert(o0_t == {1{1'b0}});
    assume(sA_t == {1{1'b0}});
    assume(sB_t == {1{1'b0}});
  end
endmodule
```

Listing 4.4: Module definition that holds assumptions and assertions for taint inputs and outputs. It has the same port definitions as the wrapper module.

```
module simple_bus_tb(
    input i0,
    input i0_t,
    input i1_t,
    input i1,
    output o0_t,
    input sA_t,
    input sA,
    input sB_t,
    input sB
);
    simple_bus DUT (i0,i0_t,i1,i1_t,o0_t,sa,sa_t,sb,sb_t);
endmodule
```

Listing 4.5: Top module created by our script, which connects the verification IP to the wrapper module. A DUT is instantiated and the verification IP is connected to it. The only output is the taint output that we monitor for tracking information flow.

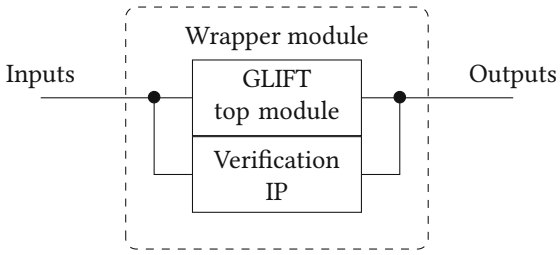


Figure 4.3: We connect two modules directly to each other. Our created module has the same port specification.

in Listing 4.6. It controls parameters such as the verification engine and its options. In its script section, input files can be prepared to match the needs of the verification such as loading the top module and its submodules. The files section needs to state all input files.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
[options]
    mode bmc
    depth 5
    expect pass

[engines]
    smtbmc

[script]
    read -formal IFT_example.v
    read -formal IFT_example.sv
    verific -import IFT_example_tb
    prep -top IFT_example_tb

[files]
    IFT_example.v
    IFT_example.sv
```

Listing 4.6: Symbiyosys configuration example: the configuration file is created based on the test case. It is splitted in sections. Output files, generated by the *pattern graph specification language information flow tracking (PGSLIFT)* script, are loaded and prepared using normal Yosys commands as Symbiyosys is a Yosys frontend.

4.4 Tool chain

Our method follows a simple flow as presented in Figure 4.4. We split the implementation into three distinct tools:

1. GLIFT generation, as a plugin for Yosys,
2. PGSLIFT script written in Python,
3. VERIFICATION script written in Python.

Our provided Python scripts support Python version 3.7 and greater. Figure 4.4 shows the verification flow and the used

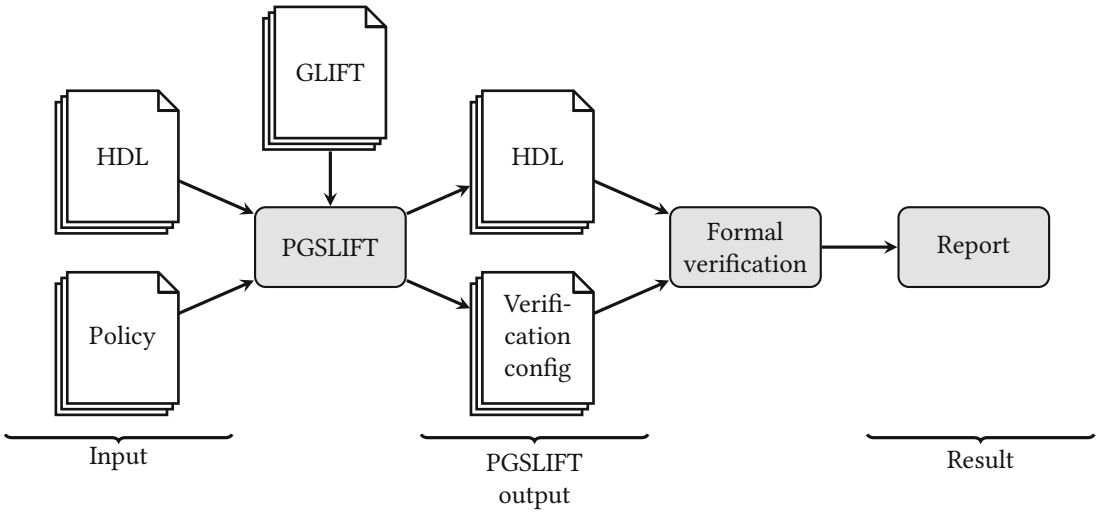


Figure 4.4: Our proposed PGSLIFT tool chain. The design and the PGSL pattern are inputs to the script. The GLIFT pass is loaded as a plugin for the synthesis tool Yosys. Output files are created for each identified test case. The verification script reads those files, runs SymbiYosys and can alter the test case files. A report is created afterwards.

tools. The PGSL pattern and the wrapped hardware design are the only inputs that the verification engineer needs to supply. The PGSLIFT script extracts test cases and then creates shadow logic and a verification configuration file for each test case. The verification script then runs the verification and creates a report. We constructed it in an interactive way. It scans the output files created by the PGSLIFT script and allows selecting which test case shall be verified. It analyzes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the result of the verification process that is stored in a smt file. This allows identifying more than one information flow since the verification tool would always finish when an information flow is found.

4.5 PGSLIFT example

This section provides an example of how our method is used to verify the flow of information in a simple hardware design featuring multiple security domains. We define IFT and describe a relation to hardware security. Based on the example design, we define security domains and use them to create security policies. These policies will then be used to verify the design using formal verification and exposing unintended information flows. After, we modify the example design in order to prevent the information flow in an unintended way.

Scenario

A microprocessor uses an external encryption module which is connected via the *serial peripheral interface (SPI)* bus. It can pass data and a key to the encryption module. A coprocessor, connected via the *universal asynchronous receiver transmitter (UART)* bus, reads the data unencrypted for monitoring purposes. Both output registers are connected to the internal registers using a simple, one bit bus. We therefore name this design *simple-bus*. In order to increase security, the UART device must never read the key.

Figure 4.5 shows a graphical representation of the corresponding design. Two registers, `Reg_Key` and `Reg_Data` provide data and key as inputs to the switching circuit. Registers `Reg_UART` and `REG_SPI` store the output data that can be accessed by the UART and SPI modules.

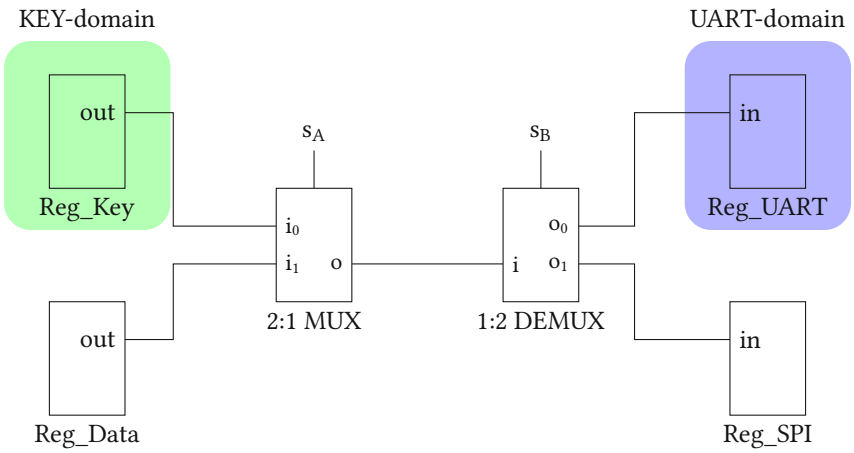


Figure 4.5: Example design without IFT logic. A leak of the key to the UART-register can not be detected automatically.

The encryption unit is connected via the SPI bus. A second output unit, which uses the UART register, should only be able to access the data register, but never read the key register. Therefore, the key and the data need to be transferred to the SPI register. Both registers for key and data are connected to a 2:1 MUX, which is then connected to a 2:1 DEMUX whose outputs connect to the SPI and UART registers. This demonstrates a typical confidentiality problem. It is not allowed to not write from a security domain with a high level to one with a lower level. For demonstration purposes, the registers will be replaced with simple inputs and outputs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Defining security domains

As a first step we define these four security domains:

- **KEY** The key for encrypting data,
- **DATA** The data to be encrypted,
- **UART** Registers that belong to the UART module,
- **SPI** Registers that belong to the SPI unit.

Defining these security domains allows implementing IFT to track any possible flow of information between the key register and the UART module. By definition, a key shall never be transferred to the UART module. Therefore, no information flow from the KEY domain to the UART domain is allowed. This verbal definition can be informal written as

$$\text{Forbidden} := \text{KEY} \mapsto \text{UART}. \tag{4.5}$$

In this simple showcase, it can be seen that the state $S = (s_A, s_B) = (0, 0)$ violates this rule. Generally, design flaws as seen here may not be detected easily, especially in bigger designs. We are able to detect an information flow from the KEY register to the UART register using IFT. To enable our tool chain to analyze the design we need to add information about the security domain to the design. This is done using simple Verilog attributes as shown in Listing 4.7 in either the source code of the DUT itself or in a separate wrapper module that connects to the DUT as shown in Chapter 2 Motivating example. Following our proposed flow, we introduced a wrapper module that adds security domains to the design. Ports without security domains, in this example the inputs s_A and s_B, can get assigned with any security domain. We chose "none" to explicitly show that it is a domain that is of no interest for our flow tracking.

```
1 (* sec_domain = "KEY" *)  input i0;
2 (* sec_domain = "DATA" *) input i1;
3 (* sec_domain = "UART" *) output o0;
4 (* sec_domain = "SPI" *)  output o1;
5 (* sec_domain = "none" *) input s_A;
6 (* sec_domain = "none" *) input s_B;
```

Listing 4.7: Definitions of security domains by using Verilog attributes

Information Flow Tracking

In order to track the flow of information the existing design is enhanced with the so-called shadow logic as described in Section 3.3 GLIFT. This additional logic allows to monitor the flow of information from the input of a design to the output. For every input bit, we add an input bit that resembles a *taint* input. This taint input marks the presence of information on the corresponding original signal bit. Additionally, a *tainted output* is generated for every output signal bit which indicates the information flow on the output. Figure 4.6 shows how both MUX and DEMUX cells have been replaced with new cells that feature the additional inputs.

The 2:1 MUX is enhanced with $i0_t$, $i1_t$, s_{At} and o_t, whereas the DEMUX is enhanced with i_t, s_{Bt}, o_{0t} and o_{1t}. We now need to instrument the additional inputs on the MUX and DEMUX according to our needs. For every input signal that origins from a security domain, we instrument the corresponding tainted input to 1. All remaining tainted inputs will

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

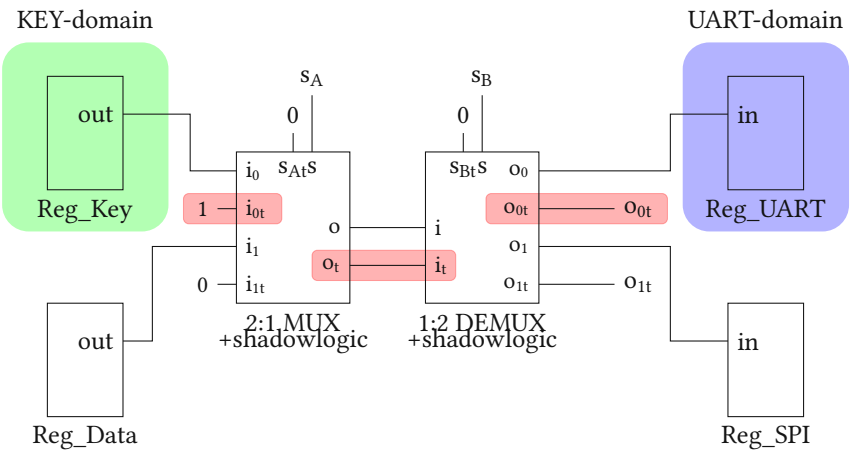


Figure 4.6: Example design enhanced with IFT logic. A leak of the key to the UART register can be detected by observing the tainted input signal.

be instrumented with 0.

On the 2:1 MUX we apply:

- $\mathbf{i_{0t}}$ corresponds to the input i_0 which is connected to the KEY register that is inside the KEY domain. Therefore, we instrument this with a logic 1.
- $\mathbf{i_{1t}}$ represents information of interest at the input i_1 which is of no interest. Therefore, we instrument it 0.
- $\mathbf{s_{At}}$ belongs to s_A which controls the MUX. It does not lay within a security domain and, therefore, we do not want to track its information flow and assign it 0.

This configuration of the 2:1 MUX with shadow logic allows to track the information flow from input i_{0t} to the tainted output o_t. In case of an information flow from the input i_0 to the output o, the tainted output o_t will be 1.

Lemma 9 *As a result, the tainted signals represent the presence of information on their corresponding signals (here: $o_t \leftrightarrow o$, $i_{0t} \leftrightarrow i_0$).*

- $\mathbf{o_t = 1}$: Information is transferred from a tainted input to the output o_t.
- $\mathbf{o_t = 0}$: No information is transferred to the output o_t.

Now the output o_t of the 2:1 MUX can be calculated as seen in the truth table in Table 4.2. The table is stripped to show only the rows where the input signals i_{0t}, i_{1t} and s_{At} are set as defined before. In this short example, it is intuitive that information from the input i_{0t} is present at the output when $s_A = 0$.

$\mathbf{i_{0t}}$	$\mathbf{i_{1t}}$	$\mathbf{s_{At}}$	$\mathbf{s_A}$	$\mathbf{o_t}$
1	0	0	0	1
1	0	0	1	0

Table 4.2: Stripped truth table of a 2:1 MUX with added shadow logic for tracking information flow. The highlighted row shows the case where information is transported from the input i_0 to the output o.

The 1:2 DEMUX is also instrumented with additional inputs:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- $\mathbf{i_t}$ belongs to the input i. Therefore, it shall be 1 when the input carries information of interest or 0 otherwise. This input will be connected to the tainted output o_t of the 1:2 MUX.

- $\mathbf{s_{Bt}}$ corresponds to the select input S_B which does not lay in a security domain and, as a result is instrumented with 0.

The tainted output of the enhanced 2:1 DEMUX can now be calculated via the truth table in Table 4.3. The output o is not modified by the shadow logic extension.

i_t	s_{Bt}	s_B	o_{0t}	o_{1t}
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

Table 4.3: Complete truth table for calculating the tainted output of the 1:2 DEMUX. The highlighted rows show the relevant cases for the given design at $S_B = 0$.

Table 4.2 and Table 4.3 can be combined to Table 4.4. By observing the tainted output o_{0t}, we notice the flow of information when o_{0t} = 1.

MUX					DEMUX				
i_{0t}	i_{1t}	s_{At}	s_A	o_t	i_t	s_{Bt}	s_B	o_{0t}	o_{1t}
1	0	0	0	1	1	0	0	1	0
1	0	0	0	1	1	0	1	0	1
1	0	0	1	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0

Table 4.4: Combined and stripped truth table for MUX and DEMUX. Highlighted columns are the select signals which can be varied by the design. The red highlighted row shows the state in which information is transferred from the KEY register to the UART register.

The output signals o_{0t} and o_{1t} show the presence of information transferred from any tainted input of this cell to its output. When wired correctly, as shown with red highlights in Figure 4.6, the output o_{0t} will be 1 in the case $S = (s_A, s_B) = (0, 0)$. Its corresponding output o_0 is connected to the UART domain and, therefore, the presence of information originating in the KEY domain at this wire conflicts with the rule stated before.

Tools

The GLIFT pass is supplied as an independent *pass* for Yosys. This means that the synthesis tool can load it as a plugin, and it can be used like any other pass.

It is loaded using the command

```
plugin -i ./gliftpass.py
```

. Shadow logic for the top module "top" can be created by running the pass by with the command

```
glift top
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

. The GLIFT pass is executed automatically by our *pgslift.py* script. Details on its implementation are provided in Chapter 5 Implementation.

PGSLIFT script

Our main script is created to perform two actions. Working as a Yosys pass, like the GLIFT pass, as well as running independently as a Python script. The PGSL pattern is handed over as an argument. The design is then prepared using normal Yosys syntax and passes are executed using the library *libyosys*. As shown in Chapter 5 Implementation, the design needs to meet some preconditions. We prepare the design using simple syntax as shown in Listing 4.8. At first, all design files are loaded. The design is then evaluated, processes are synthesized and *TECHMAP* maps the netlist to gate level cells. The last step is flattening the design in order to create a single module containing the whole logic.

```
read_verilog -sv ../designs/ift_example/mux_2x1.v
read_verilog -sv ../designs/ift_example/demux_1x2.v
read_verilog -sv ../designs/ift_example/top_secure.v
read_verilog -sv ../designs/ift_example/wrapper.v
hierarchy -check -auto-top
flatten; proc; clean; opt -full; techmap; clean; opt -full
```

Listing 4.8: Loading and preparation of the design. The result is a single module at gate level.

In order to verify the design, we need to formulate our security policy according to Chapter 5 Implementation. The final PGSL pattern results in:

```
  P0 = KEY -> UART
```

Listing 4.9: PGSL pattern as proposed and used in this thesis to describe an unallowed information flow.

This pattern is then parsed by our tool chain. The original design is analyzed and the security domains, defined as Verilog attributes, are matched with the pattern. The result of this analysis are the taint input values as seen in Figure 4.6. After that, a SystemVerilog module is created in a separate *.sv* file, which adds assertions for the given pattern as shown in Listing 4.10. All taint outputs inside the domain UART are asserted to equal 0.

```
  assert(O0_t == 0);
```

Listing 4.10: Assertion to finding information flows by monitoring the taint output.

Then, a SymbiYosys configuration file is generated, which connects the verification IP to the DUT. When run as a Yosys pass, the script is started by supplying the PGSL pattern as an argument.

```
pgslift "P0=KEY->UART"
```

Outputs are by default put in a subfolder `out/DESIGNNAME`, which resolves to `out/IFT_EXAMPLE/`. The script analyzes every test case and checks if there are matching input and output ports in the design. Only valid test cases are evaluated and shadow logic is created for them. Files for each test case are stored in a subfolder. This yields the file path created by the script:

```
out/IFT_EXAMPLE/KEY-UART/
```

Our script yields these files:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. **IFT_example.il** — synthesized design including shadow logic as Yosys internal format,
2. **IFT_example.v** — the synthesized design including shadow logic as a Verilog file,
3. **IFT_example.sby** — configuration file for the verification process,
4. **IFT_example.sv** — the verification IP including assertions and assumptions for the verification process.

Verifying the design

After running the PGSLIFT script, we can run the verification script. SymbiYosys is executed and the result evaluated. In case an information flow breaking the rules is detected, it is reported.

After running the verification tool, a report is generated to inform the verification engineer about findings gathered during the process. Listing 4.11 shows relevant parts of the generated report. The whole output of the verification software is added to the report which is cut out of this listing. The verification software then finds an information leak and reports it. The summary shows the found "positive" as a state of input values that leads to the information leak.

```
20210808_123556: Running verification for design "simple_bus" with policy "default"
20210808_123556: Running verification with command "taskset 10 sby -f simple_bus.sby"

...

20210808_123557: Finished with returncode 2, took 0.388 seconds
20210808_123557: Analyzing tracefile simple_bus/engine_0/trace.smtc
20210808_123557: Extracted state of counter example:
20210808_123557: I0:  0 I0_t: 1 I1: 0 I1_t: 0 SA: 0 SA_t: 0 SB: 0 SB_t: 0
20210808_123557: Finished verification for design: "simple_bus"
20210808_123557:
20210808_123557: ------------------------------ Summary ------------------------------
20210808_123557: ---------------------------------------------------------------------
20210808_123557:
20210808_123557: Result: Fail
20210808_123557:
20210808_123557: Found 1 unspecified information flow:
20210808_123557: Flow 1 (O0_t == 1):
20210808_123557: - I0                    :  0
20210808_123557: - I0_t                  :  1
20210808_123557: - I1                    :  0
20210808_123557: - I1_t                  :  0
20210808_123557: - SA                    :  0
20210808_123557: - SA_t                  :  0
20210808_123557: - SB                    :  0
20210808_123557: - SB_t                  :  0
```

Listing 4.11: Exemplary report of verifying the design simple-bus.

Securing the design

To ensure that information is never transferred in an unintended way, we introduce additional logic that controls the select inputs of the MUX and DEMUX gates as shown in Figure 4.7. It is controlled by the Input s_A and s_B. Its outputs are connected to the selecting inputs of both MUX and DEMUX. We name this design *simple-bus-secure*. Its truth table is shown in Table 4.5. From this we can deduce the logic functions

$$o_0 := s_A \oplus s_B \tag{4.6}$$

$$o_1 := s_A \vee \overline{s_A} \tag{4.7}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

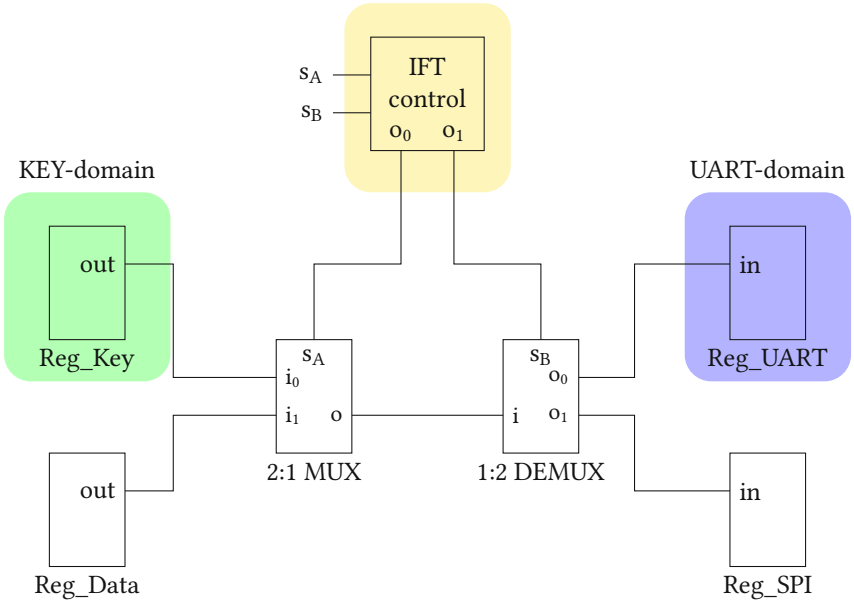


Figure 4.7: Example design enhanced with IFT logic and additional module to the control select inputs (yellow colored).

s_A	s_B	o_0	o_1
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	1

Table 4.5: Logic used for controlling the select inputs. The "insecure" state of inputs ($s_A = s_B = 0$) is not transferred to the select inputs of the original design.

The "forbidden state" $s_A = s_B = 0$ is omitted by applying o_0=0; o_1=1 to the select inputs of MUX and DEMUX cells.

Listing 4.12 shows the shortened report of verifying the secured design. The first iteration already ends with a success since no information flow could be found by the verification software.

```
20210808_123608: Running verification for design "simple_bus_secure" with policy "default"
20210808_123608: Iteration 1: Running verification with command "taskset 10 sby -f simple_bus_secure
    ↪ .sby"
20210808_123608:
20210808_123608: Output of run 1:

...

20210808_123609: Iteration 1: Finished with returncode 0, took 0.449 seconds
20210808_123609: Storing sv-file as simple_bus_secure.sv_20210808_123608
20210808_123609: Restoring sv-file simple_bus_secure.sv_bkp
20210808_123609: Finished verification for design: "simple_bus_secure"
20210808_123609:
20210808_123609: ------------------------------ Summary ------------------------------
20210808_123609: ---------------------------------------------------------------------
20210808_123609:
20210808_123609: Result: Success
20210808_123609: Verification Runs: 1
20210808_123609:
20210808_123609: Found no unspecified information flow
```

Listing 4.12: Exemplary report of verifying the design simple-bus-secure.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 5

Implementation

This chapter presents the tools used for our method as presented in Chapter 4 in every detail. We split the implementation into three distinct tools that form our tool chain:

- **glift.py** for generating *gate level information flow tracking (GLIFT)* shadow logic.
  This is a plugin for the synthesis tool Yosys and is simply named "glift". We refer to it as the GLIFT pass.

- **pgslift.py** Extracting security policies from *pattern graph specification language (PGSL)* patterns, creating test cases and output files needed to run formal verification.

- **verify.py** main script used for synthesis.
  Invokes the whole process and runs formal verification using SymbiYosys.

All tools are implemented using the programming language *Python*, version 3. Pyosys[1] wrappers allow us accessing internal data and interaction with the synthesis tool Yosys. By using Python as the programming language we have access to internal types, constructs and functions that would normally need separate integration in C++, where the core of Yosys is implemented. A wide range of available libraries can be used for example to access to the operating system or the file system directly. Furthermore, we can directly create statistics and graphs without connecting to external software. The core of Yosys is implemented in C++ and so are its prebuilt *passes*. Using Pyosys we have access to most of its internal data structure and can further ease future adaptions of our software. A downside of using an interpreted language over a compiled language is the potentially increased runtime and memory overhead. As this tool chain can be considered a prototype, we do not aim at minimizing resource consumption but prefer the advantages provided by Python.

Figure 5.1 illustrates the usage of our tool chain. At first, the design is enhanced with shadow logic. Using the PGSL pattern we analyze the modified design for Verilog attributes that introduce security domain. Output files are created that allow an automated verification run for every possible information flow between security domain. The result is saved and presented for the verification engineer to inspect.

[1] Tutzer et al., "Python Wraps Yosys for Rapid Open-Source EDA Application Development".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

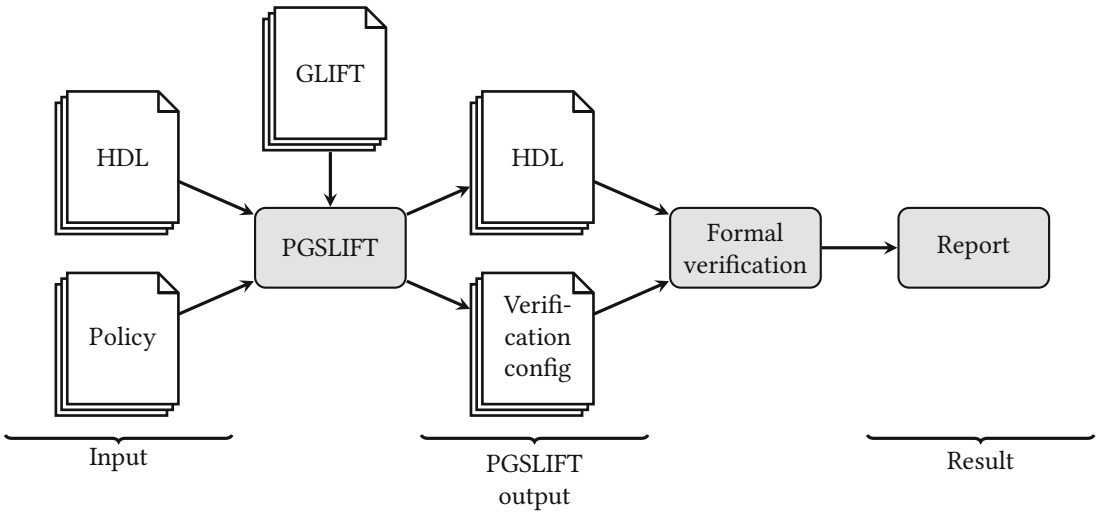


Figure 5.1: Tool chain developed to identify information flows using PGSL patterns.

Pyosys

Using Pyosys[2] we can use Python and all of its available packages and libraries without any restriction. Its functionalities and internal data structures are exploited through the *libyosys* library. This eases the use of functionalities outside the Yosys framework. For example the manipulation of external Verilog designs (*.v* format), *NumPy*[3] for faster array processing, timing methods or profiling tools to identify performance issues. The pass analyzes the given design and creates an IFT shadow logic based on the design.

In order to load a pass as a plugin, Yosys needs to be compiled with support for Pyosys wrappers. This is done by making changes to the Makefile before compiling Yosys from source as shown in Listing 5.1.

```
ENABLE_LIBYOSYS := 1
ENABLE_PYOSYS := 1
```

Listing 5.1: Changes applied to the Yosys Makefile that enable the support for Pyosys wrappers.

When developing a Yosys pass using C++, the pass has to be compiled every time in order for changes to be applied.[4] A pass created using Pyosys can be loaded as any C++ pass. Using Pyosys we are able to run Yosys using it as library *libyosys* by the Python script. Changes made to the pass can immediately be tested and debugged, since Python uses an interpreter. Although there are some downsides like increased memory consumption when using high-level Python objects compared to C++ or slightly slower execution times, the advantages outweigh the disadvantages. Running the Pyosys script using the Python interpreter instantiates the script as a main module like any other Python script like in

```
python3 pgslift.py
```

The Python file can also be used to provide the created logic as a normal pass in Yosys. We accomplish this by creating a Python class that implements the abstract class *libyosys.pass*. The pass is then instantiated by simply creating an object of this class as shown in Listing 5.2. Some functions from the super class must be implemented to make the pass work as

[2] Tutzer et al., "Python Wraps Yosys for Rapid Open-Source EDA Application Development".
[3] https://numpy.org/
[4] Wolf, *Yosys Open SYnthesis Suite.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

expected. We import *libyosys* at the beginning of the Python file as a library. Yosys loads a pass by simply importing the

```
import libyosys

def class GliftPass(libyosys.Pass):
  def __init__(self):
    super().__init__(pass_name, pass_description)
    ...
  def py_help(self):
    ...
  def py_execute(self, args, design):
    ...

p = GliftPass()
```

Listing 5.2: Basic structure to create a Yosys pass using Pyosys wrappers. Python code can be used in Yosys by defining a class of type libyosys.Pass and instantiating a pass object in code. It is executed when the pass is loaded.

Python file as a module. An object of the class *pass* is instantiated as the code outside the class definitions gets executed. This instantiation results in a call of the *__init__()* method where the pass is registered in Yosys using the *init* method of the super class. This method of the super class takes two arguments, the first being the name of the pass as a string, the second being a short description of the pass again as a string. Further methods implemented by the *Pass* class are *py_help* which is executed if the Yosys pass *help* requests a help message and *py_execute* which is run when the pass is started by Yosys.

5.1 GLIFT

As described in Section 3.3, GLIFT adds shadow logic to a hardware design based on its gate level netlist. We gain access to the netlist by synthesizing it to gate level using available Yosys commands, called *passes*.

The pass itself can be loaded using the Yosys command

```
plugin -i ./glift.py
```

It can then be used like any other pass provided by Yosys, such as the *show* or *select* passes.

Yosys data structure

Yosys internal data structure is called *register transfer level intermediate language (RTLIL)* which is inspired by *hardware description language (HDL)* terminology. As described in the manual[5] a hardware design is basically composed of these elements:

- **Design** — top object that contains modules.
- **Module** — similar to VHDL or Verilog it contains elements of a module. It contains input and output ports, cells, wires, memories. Objects can be inserted into the design by using methods provided by the module object.
- **Wire** — represents a connection between elements in the design. A wire can consist of multiple bits and each bit is connected either to another wire, a cell port or a constant driver.

[5] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- **Cell** — is the object that represents a logic function of any kind. A cell has a unique name and multiple input and output ports which are also named.
- **Gate** — a gate level cell that has limited variability compared to a Cell-type but does not need further technology mapping. Gate level cells do not need further synthesis.
- **Port** — an input or output of cells and gates.
- **IdString** — a helper object that holds a name to identify objects. This is the universal object that is used every time we use a name in RTLIL. It basically reassembles a string, but its implementation is done as a lookup dictionary to make comparisons more efficient.
- **SigSpec** — represents a signal. Basically is the low level implementation of a Wire object. It has no name and therefore no unique identifier. It links to all bits and various other data.
- **SigBit** — either a single bit of a wire or a constant (*0,1,X,Z*). Has no unique identifier as the SigSpec object.
- **Selection** — a set of modules, cells and wires. Using such a selection allows running passes on specified parts of the hardware design.

A wide range of helper objects exists that allows better interaction with the netlists elements.

Pre-conditions

The created GLIFT pass can only operate on a flattened gate level netlist. The design must therefore meet these pre-conditions:

- Processes must be translated to a netlist. The *proc* pass is used to create a netlist based on the HDL-process. Given a design D with modules m we can formulate that property as

$$\forall m \in D | !m.has_processes() \tag{5.1}$$

- The design must not contain any submodules, the number of modules must therefore equal 1.

$$\forall m \in D | len(modules) = 1 \tag{5.2}$$

  This property can be ensured by using the *flatten* pass which replaces all modules with their implementation.

- The design must only contain gate level cells c that can be analyzed and converted by the pass.

$$\forall c \in D | type(c) \in techmap.types \tag{5.3}$$

  This property is achieved using the *techmap* pass that reimplements the given logic function using only gate level cells which are `AND`, `OR`, `NOT`, `XOR`, `NAND`, `NOR`, `XNOR`, `MUX`.

- The selection must contain a whole module or the whole design. The GLIFT pass must be run on a whole module, therefore we define a property that only one module is selected and also completely. For a module m and a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Selection S we can define two properties.

$$\sum selected_modules = 1 \tag{5.4}$$

$$\forall e \in m | e \in S \tag{5.5}$$

Implementation details

The pass first checks the pre-conditions as described in Section 5.1. If the option *"-auto"* is set, the pass automatically runs passes that prepare the design, see Section 5.1 for details on available options. The current system time is stored to calculate the execution time afterwards. References of wires and cells design are stored for later access. Lookup dictionaries are created that help to find connected wires and cells. For each wire in the original design, a new wire is created that carries the corresponding taint information. Taint wires are connected equivalent to the original wires. Using a loop all cells are analyzed and a shadow logic is added depending on the cell type. The shadow logic is connected to both the original and taint wires using the wires connected to the analyzed cell and the lookup dictionaries. In case the cell type is not recognized a warning is reported and shadow logic is created by simply cloning the cell and connecting it to the taint wires equivalent to the original cell. After creating shadow logic for each cell in the design it is checked for errors. Algorithm 1 illustrates this working principle using pseudocode.

```
Data: Design D, Selection S
Result: Cells c, Wires w
if check_design(D, S) == true then
    create_original_lists() → c_original, w_original;
    clone_wires();
    create_lookup_dicts();
    for cell c in c_original do
        add sh(c) to D;
        for bit b in c do
            connect_signals;
        end
    end
end
```

Algorithm 1: Pseudo code to add shadow logic to a hardware design in RTLIL.

Creating shadow logic

In Section 3.3 we presented shadow logic functions for different cell types. Using libyosys allows us to access and modify the netlist. We add gates and wires using the *.add* methods of the module that holds the shadow logic. As seen in Figure 3.6 the shadow logic function of an AND gate is reasembled using three AND gates and one OR gate. They are added using the methods *.addAndGate()* and *.addOrGate()*. Wires connecting these gates are added using the *.addWire()* method.

GLIFT memory cells

Flip-flops and other memory type cells have no specific shadow logic function. Their functionality needs to be transfered to the shadow logic though. To include them in shadow logic, we simply create a clone of these cells. The data input

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

and output are connected to their corresponding taint values while clock, reset nand other inputs are connected to the original wires. To support all flip-flop types, we simply create a new cell object with the same type as the analyzed cell. All ports of this object are connected to the taint bit variant of the original cell port bits. Figure 5.2 illustrates this process.

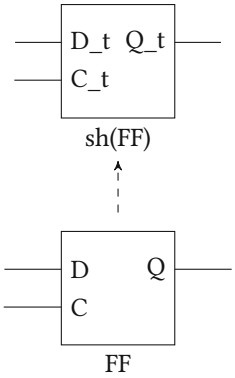


Figure 5.2: The shadow logic of a memory or flip-flop cell is equivalent to its original function. It is therefore simply cloned into the shadow logic.

Unknown cell or gate types

Our pass supports all gate types produced by the *techmap* pass of Yosys. We check the type by accessing the *.type.str()* attribute of the cell or gate. We test this by testing if the name of the cell type contains *"_dff_"*. A warning is immediately reported. To still generate an output, the unknown cell is simply cloned to the shadow logic as is done with flip-flops.

Connecting signals

In order to connect shadow logic functions the pass analyzes the netlist and connects the taint bits according to the original connection. When output o of cell C1 is connected to input i of C2, the corresponding Output o_t of $sh(C1)$ needs to connect to Input io_t of $sh(C2)$. Precise shadow logic as shown in Section 3.3 also analyzes the original input value of a cell. Therefore, the original signal is also connected to the shadow logic. Figure 5.3 illustrates these connections. Using the lookup dictionaries enables us to quickly identify corresponding signals that need to be connected.

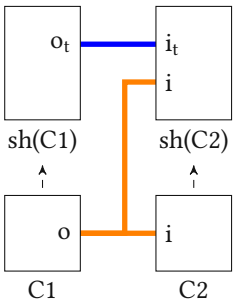


Figure 5.3: Shadow logic is created for both cells $C1$ and $C2$. The newly created taint output of $sh(C1)$ is connected to the taint input of $sh(C2)$. Depending on $C2$'s type, $sh(C2)$ could also need the signal of the original input connected. Of course the shadow logic of $C1$ also has inputs and outputs that are not shown in this figure.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Attribute propagation

As shown in Chapter 4 Methodology, we add Verilog attributes to the designs inputs in order to introduce security domains. The process also adds inputs and outputs to the design. They are assigned to the same security domain as their original counterparts. We maintain this domain information by cloning wire elements including all attributes. We can access the attribute in RTLIL using the *get_string_attribute(IdString)* method implemented by any RTLIL object that implements the *AttributeObject* interface.

Options

We use the *argparse* library to parse system arguments provided when the pass is executed. It is the "the recommended command-line parsing module in the Python standard library" [6] as described by the Python Software Foundation.

When Yosys runs the pass, it calls the *py_execute* method and supplies the arguments to it. To control the pass we feature these options:

- **-c, --conservative** creates conservative shadow logic as shown in Section 3.3 which results in less created overhead but imprecise shadow logic.
- **-v, --verbose** enables verbose mode which prints all available information gathered during the process to the output.
- **--auto** runs predefined preparation passes before creating shadow logic.
- **-h, --help** display a help message that gives a short introduction and shows all available options.
- **-d, --debug** prints additional debug information on created shadow logic.

The *argparser* object prints a help message when options *-h* or *–help* are given and ends the execution of the pass afterwards. Additionally, the *help* pass displays the help message that is returned from the *py_help()* method.

Lookup dictionaries

Yosys stores a representation of a hardware design in its internal netlist format RTLIL. It provides access to the netlist through RTLIL objects. It is basically a set of objects that have references to each other. This set of objects represent the actual netlist. During our tests we identified downsides resulting from this representation form of a netlist. Attributes of an object are often stored as a string which results in high memory consumption at designs with a high number of such objects. Also, when new objects are created by cloning, a string attribute that holds a reference to the original object is created.

Internal methods rely on iterating over the design. An example is the *select* pass. Its goal is to create a so-called selection by picking parts of the design that meet user given criteria. This is done by iterating over every element in the design and comparing its properties to those specified. Matched elements are stored in a selection which is basically a list of elements. For every specified property another iteration is executed. For example the synthesized test design AES128-T100 consists of $345,568$ cells and $1,060,497$ wire bits. The *select* pass internally uses a for-loop to iterate over all elements in the design and then returns the matching element. A test showed that selecting a random element using

[6] https://docs.python.org/3/howto/argparse.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the *select* pass takes $30.86ms$. Selecting every element in the design one by one would therefore take $13,080.67s$ which is $\sim 218min$. A solution we reasoned, and is already done in some existing passes, was to create lookup dictionaries that may take more space in memory but provides quick access when the dictionary key is the property of interest. In such a dictionary, elements are stored with a hashed key that allows for fast access. Further, creating such a lookup dictionary can be done by iterating over the elements only once, yielding a time complexity of $\mathcal{O}(n)$. By investigating the algorithm we identified the need to create multiple lookup tables as seen in Table 5.1. We managed to implement all lookup dictionary creations in linear time. They are created at the beginning of the GLIFT-pass.

Key	Value	execution time | complexity	name
Wire	List(Wire)	$\mathcal{O}(n)$	wire_to_wire_lookup
Wire	Wire	$\mathcal{O}(n)$	w_dict

Table 5.1: Lookup dictionaries created by the pass.

Hurdles

During the implementation we encountered different hurdles. Problems with Pyosys used for mapping Yosys internals to Python were discovered. We managed to overcome them in different ways. We describe all hurdles that came up during the implementation in Appendix A.3.

Reducing Design Complexity

The execution time depends on the design's complexity as shown in Chapter 6. We use the number of cells and wires in the design as a measure for its complexity. In order to perform formal verification on information flows we add additional shadow logic to the design. This increases the complexity of the design. Shadow logic is created for each cell in the original design and depends on the cell type. As a proof of concept we first featured only AND and NOT cell types. In order to create a design consisting solely of these types we converted its netlist to an *and-inverter graph (AIG)* using Yosys' *aigmap* pass. This resulted in an increased design complexity. We reduced this overhead by creating shadow logic for a larger set of allowed cell types. Yosys' *techmap* pass performs *technology mapping* and converts the netlist to gate level featuring a larger number of cell types compared to aigmap. We used findings from Section 3.3 to create shadow logic templates for all supported cell types.

5.2 PGSLIFT

This tool uses Yosys and the GLIFT pass to create output files which we use for verifying a hardware design using SymbiYosys. As shown before, Yosys can be run by running a Python script that imports libyosys. That means we execute passes and access RTLIL objects from within Python. The tool itself is executed using The functionality is

```
python3 pgslift.py
```

implemented in a pass like style so in future it will be possible to modify it and run as a distinct Yosys pass. The hardware design is provided using a synthesis script similar to that used by Yosys and documented in the manual.[7] The script is read line by line and executed using libyosys. Comments are ignored and marked by a # character.

[7] Wolf, *Yosys Open SYnthesis Suite.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Introduction

In order to define security policies we chose to use the language PGSL as shown in Section 3.4. It was designed specifically for hardware security verification and malware detection. Our main goal was to use PGSL to specify a policy that can be used to verify information flow from one security domain to another. We identified a way to introduce abstract security domains consisting of a name and a level. They are assigned to inputs as well as outputs of the hardware design. In the process of verification, the engineer creates a wrapper module that connects to all input and output ports of the *design under test (DUT)*. The manually created wrapper ensures all input and outputs are connected and maliciously added ports can already be found at the wrapper creation stage. The assignment of a security domain is done inline when writing the wrapper. Security models are linked to a hierarchy of security levels as shown in Section 3.2. In Chapter 4 we introduced these policies that describe an order of security domains. Since every input and output is assigned with a security domain label we can later precisely track the flow of information between every possible input and output combination. The *pattern graph specification language information flow tracking (PGSLIFT)* script operates on the wrapper module which is written by the verification engineer as shown in Section 2.2. It analyzes the attributes of both input and output ports. A PGSL pattern is parsed security domains and extracts an information flow policy.

Security policies

Security policies form a valid PGSL pattern. Although, we only use a subset of the PGSL syntax. This allows us to parse the pattern using string manipulation without the need for complex state machines. The pattern is passed to the tool as a string argument. We model such a pattern using a regular expression as shown in Listing 5.3. Its components are described in Listing 5.4. We use it to both check the policy for formal correctness and extracting the domain hierarchy. Separation is done by using block labels *policy_label* and *domains* which allow to split the pattern string. Using the

```
((?P<pattern_label>[\w\d]+)=)?(?P<domains>((([\w\d]+)|(->)?){3,}))
```

Listing 5.3: Regular expression used for parsing the PGSL pattern.

```
((?P<pattern_label>[\w\d]+)=)?          := optional pattern label
(?P<domains>((([\w\d]+)|(->)?){3,}))    := domain list
(([\w\d]+)|(->)?){3,}                   := either a string or a "->"
```

Listing 5.4: PGSL pattern elements split by blocks.

library *re*, featured in the Python distribution, we parse such a pattern in three steps.

1. Use the regular expression shown in Listing 5.3 to check the policy for formal correctness. An incorrect policy is detected when the *re.match* method returns an error.

2. Use regular expression *labels* to extract the policy name and the domain order. This is done by using *re.groupdict()* which creates a dictionary with the block label as key and the block itself as value. This produces to objects:

   - A string that represents the policy label,

   - A string that holds the security domain names and their connection.

3. To get a list of security domains we use Python's *str.split("->")* function to create a list of the domain names as well as their hierarchy since the resulting list is ordered by occurrence of the domain names.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

This domain hierarchy is then used to create a set of possible test cases. By matching this set with security domains found in Verilog attributes of input and output ports we can eliminate test cases without relevance.

Data: Inputs I, Outputs O, Policy P
Result: test_cases[] T
domain_list := parse_policy(P);
T := get_permutations(domain_list);
for *t in T* **do**
  for *i in I* **do**
    for *o in O* **do**
      if *not (i in t.input and o in t.output)* **then**
        remove t from T;
    end
  end
end
return *T* ;

Algorithm 2: Potential test cases are extracted and matched with the set of security domains found in input and output ports.

Security Domains

Security domains are defined using Verilog attributes. We extract this domain information of a port with the provided *get_string_attribute()* method of the *wire* class. When using the parameter *sec_domain* it returns the domain name as a string.

```
domain_name = port.get_string_attribute("sec_domain")
```

Domain hierarchy

We extract a domain hierarchy by analyzing the provided patterns. Referring to the policy from Listing 4.2 we extract the information that domain KEY is higher than UART. This information is stored in a list as shown in Listing 5.5. The order of the domain names in this list represents their relative security level. The first element in this list is therefore the element with the highest level, the second is lower than the first and so on. Information flow is only allowed from lower to higher domains, in case of list indexes that means that information can only be transferred from a domain with security level i to a domain with security level $\geq i$. In a geometrical view we can express this as:

Lemma 10 *"In this security policy information is only allowed to flow from right to left".*

```
domain_list: ['KEY', 'UART']
```

Listing 5.5: List to store domain hierarchy.

Segmenting the design

In order to reduce verification time we identified the number of cells and wires in the gate level netlist as a limiting factor. We identify the possibility to reduce verification time by using a *Divide-and-Conquer* approach. Therefore, we segment the design into smaller, independent designs. Segmentation is done in order to reduce

- Design complexity — number of elements in the design,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Verification time — execution time of the tools involved,
- Resource consumption — memory and *central processing unit (CPU)* usage of the test system.

The design complexity exponentially influences resource consumption and verification time. Segmentation means that we reduce the design's complexity by removing parts not affecting the result of the verification and are therefore insignificant for the solution. Those parts need to be identified and then removed leaving a design with reduced complexity. The set of removable parts differs between test cases. Therefore, this process is done for every test case before it can be verified. We implemented two different variants of segmentation in order to compare and find the most effective variant in terms of execution time. For comparison, we also verify the design without partitioning to create a reference. We refer to the example presented in Section 4.5 to visualize our methods.

No segmentation

A typical verification flow for this problem is shown in Figure 5.4. Design complexity is not changed, but there is no additional computation needed to segment the design.

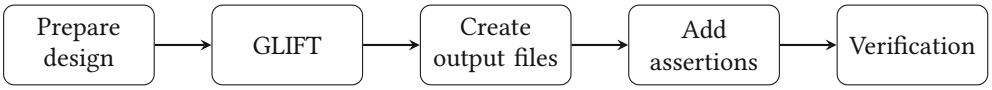


Figure 5.4: Verification flow without segmentation.

Segmentation first, GLIFT later: SFGL

Here, we segment the design before creating shadow logic. We remove input and outputs not in the security domains of interest. After the creation of shadow logic, the original output ports are removed. Yosys' *delete* pass then creates a design that consists solely of the input cone of the taint output of interest. This way all inputs not influencing the output are eliminated in hindsight. In order to formally verify the design we are only interested in the value of the taint output UART_t. The actual value of UART is insignificant. We can therefore eliminate the original output UART leaving the shadow logic output UART_t the only output of the design. Listing 5.6 shows the steps needed for segmenting the

```
delete -output o:SPI
glift
delete -output o:UART
```

Listing 5.6: We segment the design by deleting outputs that are not needed in order to verifying for information flow.

design. At first, all outputs that are not in the output domain of interest are deleted. In this example, we delete the output SPI. Then the shadow logic is created using the GLIFT pass. At the end the original output, that still has the normal function, can be deleted since we are only interested in the value of the taint output. Figure 5.5 illustrates this method for an arbitrary number of n test cases.

GLIFT first, Segmentation later

Another method is to create an IFT implementation before creating segments. Figure 5.6 It is similar to the method before but leads to different execution times as shown in Chapter 6.

1. Create GLIFT implementation and remove original output ports. This design is saved for later use.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

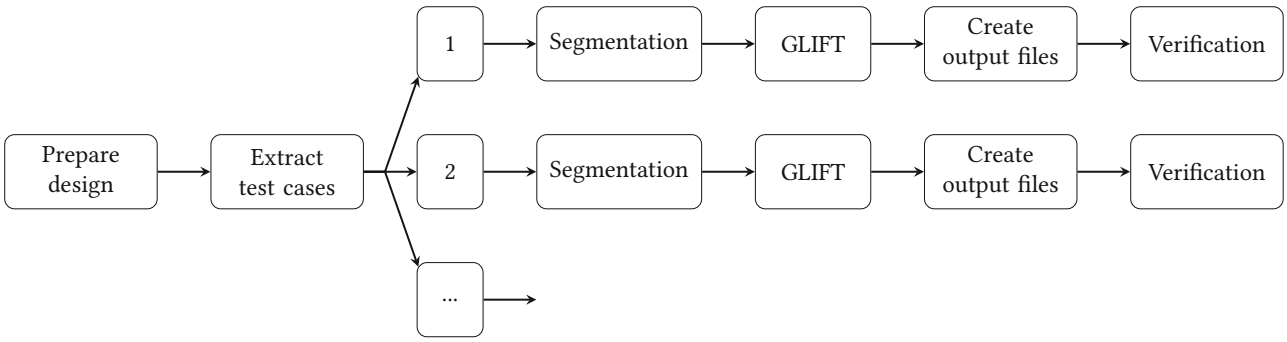


Figure 5.5: Verification flow "segmentation first, GLIFT later" — *SFGL*.

2. For every test case delete all not needed outputs and run the clean pass to have the input cone of the taint output remaining.

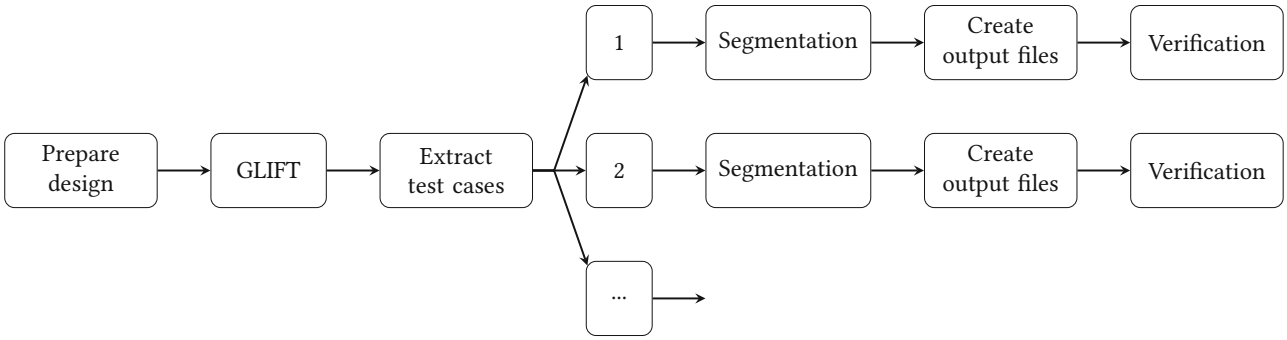


Figure 5.6: Verification flow "GLIFT first, segmentation later" — *GFSL*.

SystemVerilog assertions

Using SystemVerilog assertions and assumptions, the verification tool gets knowledge on how to instrument inputs and outputs. Formal verification is a method to test a hardware designs for such policies. They are formalized using SystemVerilog *assertions*. Assertions are constructs that we and the formal verification tool expect to be true at all times. The verification tool is able to test the design against these assertions. Listing 5.7 shows an assertion that requires the output *out_t* to equal 0 at all times.

```
always @(*) begin
  assert (out_t==0);
end
```

Listing 5.7: Assertion needed for formal verification.

Verification tools such as SymbiYosys try to find a sequence of input values in order to break these assertions. We can use *assumptions* to predefine input values. If such a sequence is found that breaks the assertions, the verification fails and a counterexample is returned. This enables us to preset the taint input values according to the design.

Lemma 11 *We prove that no information flows from an input i to an output O when the value of the corresponding output O_t can be proven to equal 0 while the taint input i_t equals 1 and all other input values and all times.*

Referring to the example design from Section 4.5 we define assertions and assumptions as shown in Listing 5.8. We assert the output $o0_t$ to equal 0 while the taint input $i0_t$ is 1. The *SystemVerilog assertion (SVA)* statements are covered

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

by an *always* block in order to mark them valid at every point in time.

```
always @(*) begin
  assume(i0_t == {1{1'b1}});
  assume(i1_t == {1{1'b0}});
  assert(o0_t == {1{1'b0}});
  assume(sA_t == {1{1'b0}});
  assume(sB_t == {1{1'b0}});
end
```

Listing 5.8: SystemVerilog assertions used to verify the example design for information flow from input i0 to output o0.

Creating output files

After we created designs that can be used for formal verification we need to create output files that automate the verification. Yosys has its own file format for storing netlists from its RTLIL. It's called *ILANG*. Those files are stored with the ending *.il*. We export the *design under test (DUT)* in ILANG format to improve write and read times since no time-consuming parsing needs to be done as would when using Verilog.

As shown in Chapter 4 we create a top module and a verification *intellectual property (IP)*. To ease the implementation, both are defined in the same *.sv* file. The top module connects the DUT to the created verification IP by instantiating and connecting them. The top module is saved as a *.sv* file. Creating the top module is done by analyzing the design using libyosys. Using simple string assembly methods we create a text file that fully complies with SystemVerilog and can be imported by the verification tool. The resulting top module consists of the module definition including inputs and outputs as well as an instantiation of the DUT and the verification IP. Last, the creation of a SymbiYosys[8] script used for formal verification is done. Again, we analyze the design using libyosys and create a configuration file needed to verify the design. The structure of such a configuration file is described in Section 5.3. In order to maintain a structure when verifying multiple designs, we decided on a directory structure as shown in Listing 5.9 where *ROOT* is the execution folder of the script. The root directory can be changed using the provided *command-line interface (CLI)* options. *TOP_-MODULE_NAME_1* is the name of the top module of the DUT. The *TEST_CASE* name is constructed using the input domain and the output domain like KEY-UART.

```
ROOT
|
+-- TOP_MODULE_NAME_1
| +-- TEST_CASE_1
| | +-- TOP_MODULE_NAME.ilang
| | +-- TOP_MODULE_NAME.sv
| | +-- TOP_MODULE_NAME.sby
| +-- TEST_CASE_2
|
+-- TOP_MODULE_NAME_2
  +-- TEST_CASE_1
```

Listing 5.9: Directory tree for saving output files.

[8] Wolf, *SymbiYosys*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Command line options

Options are processed the same way as in the GLIFT pass as shown in Section 5.1. As shown in Figure 4.4, the PGSLIFT tool requires these inputs:

- Design files,
- PGSL pattern describing the domain hierarchy.

We provide and use these options:

- **-o** define a root path to put the created files in. Will overwrite existing files or directories.
- **-v, --verbose** enables verbose mode which prints additional information gathered during the process to the output.
- **--sby** overwrites the preset SymbiYosys *options* section in the configuration file. The options are provided as a string including line breaks after each option.
- **--engine** lets the user choose the verification engine to be used. By default, *smtbmc* is used.
- **--depth** sets the induction depth. By default, the depth is 5
- **--mode** selects the segmentation mode using a number in range $0 - 2$. Segmentation methods are described in Section 5.2. This option is set to 0 by default.
  0. no segmentation
  1. SFGL — segmentation first GLIFT later
  2. GFSL — GLIFT first segmentation later
- **--no-sv** disables the creation of SystemVerilog assertions and assumptions. Useful for creating output files where they will be inserted manually afterwards.
- **--script** path to a script that loads design files. The script file is executed line by line. Each line is passed to Yosys and executed. Empty lines are ignored as well as comments after a "#" character.
- **-h, --help** display a help message that gives a short introduction and shows all available options.
- **PATTERN** The pattern is implemented as a positional argument and therefore does not need an argument name.

Time measurements

We measure the execution time by storing the current system time before and after every critical step in order to calculate absolute execution times. Python's timing library *time* is used to get the current system time using the function *time.time()*. We measure all relevant times during the execution and create statistics for evaluating and comparing results as presented in Chapter 6.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

5.3 Formal verification

To identify the absence or presence of information flows we use formal verification as presented in chapter 3. The verification process can be done either manually or with our provided tool *verify.py* written in Python3.

We use the tool SymbiYosys which is a frontend to the used synthesis tool Yosys. That means that it uses Yosys and extends its functionality. It is controllable via a configuration file.

Configuration file

The configuration file is stored with the extension *.sby*. The file itself is a simple text file that is split into different sections. It controls every aspect of the verification process such as the used formal verification engine, the number of verification steps and also Yosys passes to run on the design files. The engine to be used is provided under the *[engine]* section. Configurations such as the engine mode or the induction steps are provided under the *options* section. To select a hardware design we must provide a short script consisting of one or more Yosys commands that load the design and, in our case, adds the assertion file and the top module. The last section *[files]* lists all source files that are needed for the verification process.

Verification process

The verification process is started by executing SymbiYosys using the command

```
sby -f TOP_MODULE_NAME.sby
```

where TOP_MODULE_NAME is the name of the designs top module. The option *-f* allows the verification tool to overwrite already existing output files created by previous verification attempts. Alternatively, the option *-b* as in *sby -b *.sby* creates a backup of the found verification files. After executing the script the verification engine is run with the options provided in the configuration file. Output files are created in a directory that has the same name as the top module of the design. Both SymbiYosys and the verification engine create various output directories and files containing intermediate files created during the verification and various log files. In case the verification process fails the verification tool produces a waveform file that shows the counterexample as a state of inputs that lead to the fail. The return code implies whether the process succeeded or failed. From this we can deduce wether an information flow was identified or not.

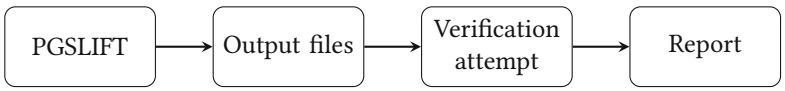


Figure 5.7: Verification process create a report. The HDL and verification configuration are used to execute the verification tool. Its output is analyzed and a report created. In case the verification fails the output files of the verification tool are analyzed to provide information on the input state that lead to identifying an information flow.

Verification helper tool

To control the whole verification process we created the tool *verify.py*. It reads the created files from the PGSLIFT tool, starts the verification process and evaluates the result. All findings and outputs of the used tools are stored in a log file.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

The tool follows the flow diagram as shown in Figure 5.7. All steps are automated after the user chooses the correct test design and test case. The first step is the selection of the target design based on the outputs created by the former tool PGSLIFT. It does so by analyzing the output directory *out*. Every directory contains a different test design. For each test case extracted from the design there is a separate directory containing source and configuration files. File access is eased by using the library *os* which is part of the Python distribution. Using the *glob* library we are able to scan the directories with less operating system calls. The user is provided a list of found designs and then gets presented a list of included test cases. After selecting the test case to be verified, the tool analyzes the source files in the directory associated with the test case. All files that may be altered during the process are backed up using the library *shutils*. Then the verification process is started using Python's *subprocess* library which allows running external programs and read their return codes. If the verification succeeds, it is reported and the script ends. In case of a fail, the trace file created by the verification engine is analyzed by the tool. From this trace file we extract the input state that results in an information flow. A report that contains all information is created and presented to the user.

Command line options

The tool can be run as a Python script by executing the command *python3 verify.py*. To control the tool we provide a range of options that can be used. Similar to the other tools, we used the *argparse* library to handle system arguments.

- **-h, --help** Display a help message that gives a short introduction and shows all available options.
- **--dir** Path to directory containing source files for direct execution without interaction with the user. It is implemented as a positional argument and has no argument name.

Logging

The final result of the verification process is a report that holds all information valuable for the verification engineer. Each line of the log file begins with a timestamp. Outputs of external programs are also stored in the log file, such as SymbiYosys. Furthermore, calculated times are logged that are useful for comparing verification attempts. The log file's name is composed of `log`, a timestamp and the ending *.log*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Chapter 6

Results

In this chapter we present the experimental setup and the test methods that were used. In order to draw conclusions on the research questions presented in Section 1.3, we provide details and results on experiments. Then, we provide an overview of the chosen test designs and provide all measured results. After that, we provide details on all performed experiments and present the measurements and evaluate them. In the end, we summarize our findings discuss the results. Further, we compare our method to the available method *SecVerilog* as presented in section 3.6. We demonstrate the differences at the design phase and at verification. Time measurements are provided as long as they are statistically significant.

6.1 Methods

The execution time is expected to vary due to various effects inside the experimental setup such as caching that are not predictable. As a statistical method, we therefore execute time measurements 5 times repeatedly unless otherwise stated. Since we can not expect one of those measurements to be the minimal execution time we do not provide the lowest measured timing. The execution time can also be extended by the system prioritizing other active tasks. We want to provide an expected execution time, therefore we use the mean value of time measurements in summaries. Detailed data is provided for each test.

Experimental setup

All tests and measurements were done on a system consisting of these components:

- CPU: AMD Ryzen5 2600 6x 3.4GHz,
- Mainboard: Gigabyte B450 Aorus Pro Wifi,
- RAM: Corsair Vengeance LPX DDR4 3000MHz CL16 - 32GB,
- GPU: Sapphire R580 Nitro Pulse 8GB,
- Disk: Samsung SSD 970 EVO 500GB,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Operating system: Ubuntu 20.04.01 LTS.

We provide measurements of execution times of our tools. Further, we measured the memory consumption of our tools as well as the external tools used, such as SymbiYosys. Execution times will be measured n times to create statistics. Testing is done using the python library *timeit*. It allows to repeatedly execute methods and measures the time needed for the execution. Python's *time* library has a precision of 1 µs when running on Linux operating systems. Python v3.7 and higher introduce a new time measurement function that allows to measure with a precision of 1 ns. To maintain compatibility to older versions, we chose to use the less precise functions. Memory load is measured using the Python library *psutil* which allows to gather the memory consumption of a running process or through the software *top*.

6.2 Test designs

In order to conclude on the research questions we perform experiments on different test designs and measure relevant values. Our chosen test designs are presented in Table 6.1. At first, we perform measurements on the designs presented in Chapter 4. To compare our work to related works, we also test our method on designs that are used in other works. We investigate the number of wire bits and cells in the design, which will be called *cell and wire count*, to compare the size of these designs. These counts are measured after the designs have been prepared in order to be processed by the PGSLIFT script.

Design name	Cells	Wire bits	Inputs	Input domains	Outputs	Output domains
simple-bus	4	8	4	4	2	2
simple-bus-secure	12	19	3	3	2	2
AES128-T100	345,568	1,060,497	3	3	1	1
RSA-T100	3,251	3,911	6	3	2	2

Table 6.1: Used test designs, their cell and wire counts and the number of security domains we introduce for our tests.

simple-bus

We presented this design in Section 4.5. It features two input and two output security domains. The goal of the verification is to identify an information flow from any input in security domain *KEY* to any output in security domain *UART*. Table 6.2 shows the number of cells and wires of the original design, with added shadow logic and after segmentation. As

Configuration	Cell count	Wires	Wire bits
Original	4	7	8
GLIFT	23	30	32
Segmented	16	22	24

Table 6.2: Elements of design simple-bus.

shown in Table 6.2 the additional logic adds 19 cells and 24 wire bits to the design. Segmentation reduces this overhead to twelve cells and 16 wire bits. This is a reduction of 30.4 % in cells and 25 % in wire bits.

simple-bus-secure

This design adds additional logic to the exemplary design simple-bus. We demonstrated that using this controller module, no information flow breaks the security policy given by the PGSL pattern. As shown in Table 6.3 the additional

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

logic adds 43 cells and 55 wire bits to the design. Segmentation reduces this overhead to 36 cells and 47 wire bits. This is a reduction of 12.7 % in cells and 10.8 % in wire bits.

Configuration	Cell count	Wires	Wire bits
Original	12	14	19
GLIFT	55	64	74
Segmented	48	56	66

Table 6.3: Elements of design simple-bus-secure.

AES128-T100

The design *AES128-T100* is provided by trust-hub.org. [1][2][3] It leaks the cryptographic key to an additional output called *Capacitance*. During the creation of this thesis problems in the provided designs were found that lead to the payload of the *hardware Trojan (HT)* being deleted during logic optimization. Changes were made, so the design can be synthesized to gate level. Further, the key is not leaked to a distinct output anymore but to the default output *out* of the cryptographic core.

	Original	GLIFT added	Segmented
Wires	79,400	1,273,584	1,269,411
Wire bits	1,060,497	3,235,778	3,167,864
Cells total	345,568	1,753,304	1,752,276
AND cells	1088	837,176	837,096
OR cells	277,606	843,256	843,176
NOT cells	52,616	52,616	52,612
XOR cells	8,256	8,256	7,904
MUX cells	0	0	0
DFF cells	6,000	12,000	11,488
Memory	1,094 MB	3,784 MB	3,712 MB

Table 6.4: Elements in the design AES128-T100 in its original state, with added shadow logic (GLIFT) and after segmentation. Memory is the memory allocated by the synthesis tool with the loaded design.

```
module aes_128_wrapper(clk, state, key, out);
  (*sec_domain="LOW"*) input clk;
  (*sec_domain="LOW"*) input  [127:0] state;
  (*sec_domain="KEY"*) input  [127:0] key;
  (*sec_domain="OUT"*) output [127:0] out;

  aes_128 top (clk, state, key, out);

endmodule
```

Section 6.2 shows the wrapper of the design. From the description of the HT we extract information that the key leaks to the output. We are only interested in an information flow from input *key* in security domain KEY to output *out* in domain OUT, we therefore create the security policy shown in Listing 6.1.

```
P=KEY->OUT
```

Listing 6.1: Security policy for identifying an information leak in test design AES128-T100.

When the design is synthesized using our preparation script shown in Listing 6.6, it consists of 1,060,497 wire bits and 345,468 gate level cells as shown in Table 6.4. Adding shadow logic adds 2,175,281 wire bits and 1,407,736 cells to the

[1] https://www.trust-hub.org
[2] Salmani, Tehranipoor, and Karri, "On design vulnerability analysis and trust benchmarks development".
[3] Shakya et al., "Benchmarking of Hardware Trojans and Maliciously Affected Circuits".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

gate level netlist. Segmentation reduces this by 67,914 wire bits and 1,028 cells. The reduction is therefore 2.1 % in wire bits and 0.06 % in cells. Memory consumption of the synthesis tool with the design in memory is increased by 2,690 MB and segmentation reduces it by 72 MB which is 1.9 %. We further see that segmentation reduces this overhead by 0.58 % in cells, 14.4 % in wire bits and 1.43 % in memory.

RSA-T100

This design is also provided by trust-hub.org and originally called *BasicRSA-T100*. It is a generic cryptographic core that uses the RSA encryption and a key with 32 bit length. We chose it since it represents a valid use case for our method. The design represents a HT that leaks the key to the output when the plaintext input equals $32'h44444444$. Its implementation is split in Verilog and *very high speed integrated circuit hardware description language (VHDL)* modules. Our synthesis tool Yosys is capable of analyzing a VHDL design using the external tool *verific*. A version of Yosys featuring this external tool is available via *YosysHQ*. [4] It comes with a prebuilt version of both Yosys and SymbiYosys. This prebuilt version of Yosys does not have Pyosys support built in, therefore we can only use it so synthesize the design but not run our passes on. Listing 6.2 shows the modified preparation script to load both Verilog and VHDL design files. The output is stored in *ILANG* format, so we can process the netlist using the Pyosys enabled Yosys version.

```
read -vhdl rsacypher.vhd modmult.vhd
verific -import -all
rename RSACypher(Behavioral) top
read_verilog rsa_wrapper.v
hierarchy -check -top RSACypher_wrapper
write_ilang RSA_T100.il
proc; clean; memory; clean; opt -full; clean;
flatten
techmap
opt -full
opt_clean
clean
rename -top RSA_T100
```

Listing 6.2: Preparation script for loading and synthesizing the RSA-T100 design

Similar to the design AES128-T100 its malicious part is not synthesizable in its provided form. The design was therefore fixed and this new version used. Table 6.5 summarizes the number of cells and wires in the design. Details are provided in Table 6.6 where all cell types are compared.

Configuration	Cell count	Wires	Wire bits
original	3,244	1,060	4,397
GLIFT	26,983	21,872	29,183
segmented	21,194	16,920	23,594

Table 6.5: Comparison of cells and wires of design RSA-T100 in different states.

We see that the creation of shadow logic adds 19,950 = 653.33 % cells, 15,860 = 1596.23 % wires and 19,197 = 536.59 % wire bits to the design. By segmenting the design we delete 189 = 0.89 % cells, 1,222 = 5.18 % wire bits and 187 = 1.1 % wires.

Listing 6.3 shows the wrapper module that assigns security domains to the modules inputs and outputs.i From the specification of the HT, which is an informal description that the Trojan leaks the exponent to the cypher at a certain plaintext, we extract the PGSLIFT policy

[4] https://www.yosyshq.com/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

	Original	GLIFT added	Segmented
Wires	1,060	16,920	16,733
Wire bits	4,397	23,594	22,372
Cells total	3,244	21,194	21,005
AND cells	1,009	10,302	10,227
OR cells	606	7,136	7,086
NOT cells	94	1,762	1,762
XOR cells	520	520	520
MUX cells	556	556	524
DFF cells	459	918	886
Memory	21.53 MB	47.39 MB	67.73 MB

Table 6.6: Elements in the design RSA-T100 in its original state, with added shadow logic (GLIFT) and after segmentation.

```
1  module RSACypher_wrapper(clk, ds, reset, ready, inExp, inMod, cypher, indata);
2      (*sec_domain="LOW"*) input clk;
3      (*sec_domain="LOW"*) input ds;
4      (*sec_domain="KEY"*) input [31:0] inExp;
5      (*sec_domain="LOW"*) input [31:0] inMod;
6      (*sec_domain="LOW"*) input [31:0] indata;
7      (*sec_domain="LOW"*) input reset;
8      (*sec_domain="CYPHER"*) output [31:0] cypher;
9      (*sec_domain="LOW"*) output ready;
10
11     top top (clk, ds, reset, ready, inExp, inMod, cypher, indata);
12
13 endmodule
```

Listing 6.3: Wrapper for test design RSA-T100.

```
1  P=KEY->CYPHER
```

Other security domains are of no interest when testing for an information flow violating this policy, therefore domains of other ports can be assigned without paying attention to the verification process. We assign a LOW security domain to those ports that are of no interest.

6.3 Information leak detection

simple-bus

In Section 4.5 we showed the verification results of test design *simple-bus* and *simple-bus-secure*. From the report shown in Listing 4.11 and Listing 4.12 we see the results of the verification process from our method. Verifying the design with the unspecified information flow to an output in security domain UART resulted in the detection of an information leak. The report shown in Listing 4.11 the input state that broke the assertions, containing the select input values $S_A = S_B = 0$. After modifying the design and adding the IFT control module that avoids this state the verification passes. From that we deduce that we could not identify an information flow. The report is shown in Listing 4.12.

RSA-T100

Listing 6.4 shows a stripped version of the report generated by verifying the design RSA-T100. Here an information flow is detected. In this design there is an intended information flow from domain KEY to CYPHER since the key influences the output value of the cryptographic function. The malicious part of the design transmits the key used in the cryptographic process directly to the output. Our method is able to detect this information flow. For this test design

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

we can therefore conclude that our method detects information flowing to the output.

```
20210908_054529: ------------------------------ Summary ------------------------------
20210908_054529: ---------------------------------------------------------------------
20210908_054529:
20210908_054529: Result: Fail
20210908_054529:
20210908_054529: Found 1 unspecified information flow:
20210908_054529: Flow 1: (cypher_t == 0)
20210908_054529: - clk                  :  1'b0
20210908_054529: - clk_t                :  1'b0
20210908_054529: - ds                   :  1'b0
20210908_054529: - ds_t                 :  1'b0
20210908_054529: - inExp                :  32'b00000000000000000000000000000000
20210908_054529: - inExp_t              :  32'b11111111111111111111111111111111
20210908_054529: - inMod                :  32'b00000000000000000000000000000000
20210908_054529: - inMod_t              :  32'b00000000000000000000000000000000
20210908_054529: - indata               :  32'b00000000000000000000000000000000
20210908_054529: - indata_t             :  32'b00000000000000000000000000000000
20210908_054529: - reset                :  1'b0
20210908_054529: - reset_t              :  1'b0
```

Listing 6.4: Exemplary report of verifying the test design RSA-T100.

AES128-T100

Similar to the design RSA-T100 the design transmits the secret key to the output therefore there is an information flow from the input to the output. This information flow is detected by monitoring the output value of the shadow logic. Listing 6.5 shows an exemplary, stripped report generated by running our verification tool on this design.

```
20210908_041412: ------------------------------ Summary ------------------------------
20210908_041412: ---------------------------------------------------------------------
20210908_041412:
20210908_041412: Result: Fail
20210908_041412:
20210908_041412: Found 1 unspecified information flow:
20210908_041412: Flow 1 (out_t == 0):
20210908_041412: - clk                  :  1'b0
20210908_041412: - clk_t                :  1'b0
20210908_041412: - key                  :  128'
    ↪ b00000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000000000000000000000000000
20210908_041412: - key_t                :  128'
    ↪ b11111111111111111111111111111111111111111111111111111111111111111111111111111111
    ↪ 11111111111111111111111111111111
20210908_041412: - state                :  128'
    ↪ b00000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000000000000000000000000000
20210908_041412: - state_t              :  128'
    ↪ b00000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000000000000000000000000000
```

Listing 6.5: Exemplary report of verifying the test design AES128-T100.

6.4 Complexity analysis

In order to make a statement on expected execution times of the different tools we identify the complexity of the algorithms. We use upper bounds $\mathcal{O}()$ as a measure and compare it to the measurements. The algorithms work on the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

design and therefore, we can define their inputs as

$$c\ldots \text{ cell count,} \tag{6.1}$$

$$w\ldots \text{ wire count.} \tag{6.2}$$

We process designs synthesized to a gate level netlist, therefore the wire count is represented by the number of single bits. The synthesis tool does not create single bit wires at the synthesis, therefore the number of wires and the wire bits can differ. Therefore, the wire count is used as a synonym for the number of what is called *wire bits* in Yosys. For identifying upper bounds based on Python's internal functions we refer to its documentation[5] providing valid upper bounds.

GLIFT

The algorithm of the GLIFT pass can be split into three parts.

- Preparations where the design is analyzed and prepared.
- Creation of lookup dictionaries.
- Creation of shadow logic.

Details on the implementation and the used algorithms are found in Section 5.1. We perform a complexity analysis to draw conclusions on the expected execution times.

Preparation passes

Preparation of the design is done using Yosys passes. The source files are parsed and the RTLIL netlist is created using the commands shown in Listing 6.6. We stick to this preparation scheme for all test designs. Slight changes are necessary for importing VHDL source files, as in the RSA-T100 designs. The test design AES128-T100 uses more memory than available on the testing system. We therefore add another optimization run to the preparation script for this test design.

```
read_verilog *.v            # Load all designs sourcefiles
hierarchy -check -auto-top  # Set the top module
proc                        # Unroll processes
opt -full                   # Logic optimizations
clean                       # Remove unused cells and wires
flatten                     # Replace submodules with their implementation
techmap                     # Map cells to gate level cells
```

Listing 6.6: Preparation script to load the test designs and prepare them for shadow logic implementation with the GLIFT pass.

It is not part of this work to analyze existing algorithms, therefore we only provide measurements of the execution time of these passes. During the measurements it turned out that for designs with a high complexity, the execution time and memory consumption vary with the order that these passes are used. The complexity of the used algorithms is therefore not equal throughout the passes.

[5] https://wiki.python.org/moin/TimeComplexity

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Lookup dictionaries

Lookup dictionaries are created by iterating over the design once or twice and analyzing properties of the cells or wires. The execution time therefore is a function of the cell and wire count c and w. We calculate the upper bound of the time complexity to

$$f_{\text{lookup}} = \mathcal{O}(w), \tag{6.3}$$

$$f_{\text{lookup}} = \mathcal{O}(c). \tag{6.4}$$

Executing multiple dictionary creations consecutively does not affect this upper bound. Lookup tables can therefore be created in linear time.

Shadow logic creation

The algorithm to create shadow logic is given in Algorithm 1. Testing the design is done by accessing properties of the containing list objects and therefore the upper bound $f_{\text{check}} = \mathcal{O}(1)$ is valid. The same applies to the creation of lists for storing elements. As stated, lookup dictionary creation is of linear time complexity. The main loop iterates over all cells, having linear time complexity $f_{\text{main}} = \mathcal{O}(n)$. Adding shadow logic cells depending on the cell type is done by adding cell and wire objects to the netlist which resolves to constant time $f_{\text{shadow_cells}} = \mathcal{O}(1)$ since it's not depending on the number of input wire or cell counts. The same applies to connecting the shadow logic to the original logic with $f_{\text{connection}} = \mathcal{O}(1)$. The resulting upper bound of time complexity can therefore be calculated to $f_{\text{glift}} = \mathcal{O}(n)$. We can further state that the creation of shadow logic yields exactly one execution of the adding function for each cell in the design.

Additional complexity

The GLIFT pass has other features such as argument parsing or file system access by using Python libraries. We consider them of linear complexity for the designs cell and wire count. Therefore, it's not feasible to do a complexity calculation on all used modules.

Conclusion

Combining these findings leads to the conclusion that the GLIFT pass overall has linear time complexity $f_{\text{pass}} = \mathcal{O}(n)$ with $n = max(c, w)$.

Design complexity

We use the number of cells n and wires m as a measure for the complexity of a hardware design presented as a gate level netlist. It influences the execution time of operations done on the design $t = f(n, m)$ as will be discussed later. The process of creating and adding shadow logic to the design will be called *GLIFT pass*. We calculated the complexity of the GLIFT pass to

$$f_{\text{glift}} = \mathcal{O}(n) \tag{6.5}$$

with n being the cell count. From this upper bound we deduce that reducing the number of cells and wires can reduce the execution time. GLIFT execution time is directly dependent to the initial cell and wire count as we show in Section 6.5.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Depending on the cell type and its implementation as seen in Section 3.3 we can calculate the number of elements in the GLIFT-enhanced design.

Using this equation we can deduce Equation (6.6) that allows to calculate the cell count of the created shadow logic C_{sh} from the number of cells in the original design.

$$C_{\text{sh}} = 21 * C_{\text{MUX}} + 4 * C_{\text{AND, NAND, OR, NOR}} + C_{\text{NOT, FF}} \tag{6.6}$$

The cell count for the design added with the shadow logic C_{GLIFT} can then be calculated using Equation (6.7).

$$C_{\text{GLIFT}} = 22 * C_{\text{MUX}} + 5 * C_{\text{AND, NAND, OR, NOR}} + 2 * C_{\text{NOT, FF}} \tag{6.7}$$

We are not able to draw exact conclusions on the number of wires since the number of wires in the design is also a matter of how cells are connected to form the design's functionality as needed. Nevertheless, it is possible to draw conclusions on the internal number of wires created by the shadow logic for each cell type as shown in Equation (6.8).

$$N_{\text{sh}} = 19 * C_{\text{MUX}} + 3 * C_{\text{AND, NAND, OR, NOR}} \tag{6.8}$$

PGSLIFT

The functions of the PGSLIFT tool, as presented in Chapter 4, can be split in parts as shown in Figure 6.1. Depending on the segmentation method, the order of their execution varies.

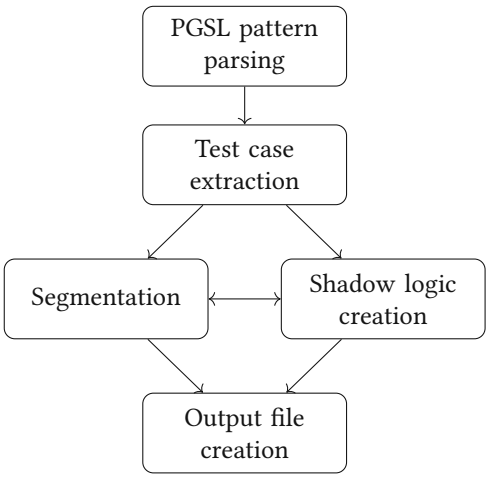


Figure 6.1: Functions of the PGSLIFT script that we analyze. Depending on the segmentation mode the actual sequence of operations varies.

Again, we analyze these operations independently and then conclude on the overall time complexity.

Regular expression matching

The PGSL pattern is matched using a regular expression. Python's *re* library is used to compile the regular expression as seen in Listing 5.3. The input pattern is then matched against this expression. Compiling the pattern can be done with complexity $f_{\text{compile}} = \mathcal{O}(2^n)$ with n being the characters in the pattern. [6] String operations to extract the domains have

[6] https://swtch.com/ rsc/regexp/regexp1.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

linear complexity for the number of characters in the strings resulting in $f_{\text{str}} = \mathcal{O}(n)$. We use the design simple-bus to provide measurements of the execution times for pattern compilation and pattern matching. As shown in Section 4.5 PGSLIFT example, the design features 4 security domains:

- *KEY*
- *DATA*
- *UART*
- *SPI*

Using the *permutations()* method provided by the *itertools* library we create a list of all possible patterns. [7] The method creates permutations with all domains included, meaning all permutations for 4 domains are 4-tuples. These are called full-length permutations. The number of permutations created by this method is calculated to:

$$\frac{n!}{(n-r)!} \tag{6.9}$$

with n being the number of input elements and r being the number output tuple length. We extend this method using a length parameter that allows to set the length of the generated tuples. Using a for loop we create permutations for all valid domain combinations. Information flows inside a domain are allowed, therefore a valid pattern describes a hierarchy of at least two security domains. Therefore, the lower bound of$r$ is $r = 2$. The number of generated patterns can therefore be calculated to:

$$\sum_{r=2}^{n} \frac{n!}{(n-r)!}. \tag{6.10}$$

With $n = 4$ this yields

$$\sum_{r=2}^{4} \frac{4!}{(4-r)!} = 60 \tag{6.11}$$

which equals the experimental result generated by our script. These tuples are processed to form valid patterns by joining the elements of the tuple using the arrow ->. The code for creating the test patterns is shown in Listing 6.7. The resulting list of patterns consists of 60 possible patterns that describe all possible combinations of input and output domains. We expect the number of security domains and therefore test cases to be low in any design of interest. The

```
domains = ["KEY", "DATA", "UART", "SPI"]
patterns = list()
for l in range(2,len(domains)+1):
    patterns.extend(list(map(lambda x: "->".join(x), itertools.permutations(domains, l))))
```

Listing 6.7: Pattern generation for all test cases given a unsorted domain list.

resulting time is therefore expected to be lower than the execution time of the GLIFT pass. Table 6.7 shows the compilation times of the pattern. The pattern is compiled once when the tool starts, therefore we can consider it constant for the execution time of the tool. We see that the time needed to compile the regular expression (regex) is $\sim$ 1.1 µs. Compilation times of the PGSL pattern depend on the length of the pattern, therefore we measure execution times repeatedly for each case. We provide the maximum and the minimum of these measurements. This also applies to the matching time, where the compiled PGSL pattern is matched with the regular expression.

[7] https://docs.python.org/3/library/itertools.html#itertools.permutations

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Measurement	Value	Unit
Regex compilation mean	1.095	µs
Regex compilation std dev	0.767	µs
Compilation means max	78.452	µs
Compilation means min	0.797	µs
Compilation std dev max	45.671	µs
Compilation std dev min	14.290	ns
Matching means max	6.427	µs
Matching means min	2.224	µs
Matching std dev max	4.102	µs
Matching std dev min	0.251	µs

Table 6.7: Execution times for compiling and matching the regular expression and patterns.

Test case extraction

Test case extraction is done by creating all possible tuple of found security domains and matching them with existing input and output security domains. This is done using the *itertools* library that features a *permutations()* method that operates on the list of extracted domain names. From its implementation [8] we can deduce its time complexity being $f_{\text{permutations}} = \mathcal{O}(n)$. Creating a representing test case object and storing it in a list is done by iterating over the list of possible test cases with $f_{\text{extract}} = \mathcal{O}(n)$ with n being the number of valid test cases.

Segmentation

We introduced three segmentation modes.

- No segmentation,
- Segmentation first GLIFT later (*SFGL*),
- GLIFT first segmentation later (*GFSL*).

No segmentation creates shadow logic on the whole design without segmenting it. There is no time complexity for us to identify since it is just an execution of the GLIFT pass. *SFGL* and *GFSL* are described in detail in Section 5.2.

The intention of segmentation is to create the input cone of the output selected for verifying its taint value. We could use Yosys's *select* pass to create this input cone. As discussed in Section 5.1 it works by iterating over the design and selecting objects depending on their properties in each iteration. Extracting the input cone of a given output results in a time complexity $f_{\text{input_cone}} = \mathcal{O}(n^2)$. Our segmentation of a design is done by identifying inputs and outputs irrelevant for the verification result and delete them using the *delete* pass. We then use Yosys's *clean* pass to automatically remove unattached cells and wires leaving behind the input cone of the given output.

We did not analyze the algorithms used in these passes, instead we analyze their execution time in order to draw conclusions on the execution time. Analyzing the algorithms could be done in future works by reverse engineering the algorithms from the C++ implementation.

SFGL Based on the extracted test cases segmentation is done. This reduces the cell and wire count from n to m with $m < n$. After that, the GLIFT pass is executed on this reduced design giving us the upper bound $f_{\text{SFGL}} = \mathcal{O}(m)$ as shown in Section 6.4. Shadow logic is created for the outputs relevant for the verification result. Since the design was

[8] https://docs.python.org/3/library/itertools.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

segmented before, the original output is still present in the design. To improve verification time original outputs need to be removed again using *delete* and *clean* passes.

GFSL Here, shadow logic is created before deleting any element from the design therefore resulting in $f_{\text{GFSL}} = \mathcal{O}(n)$ with n being the cell count of the original design. This introduces shadow logic also for cells not being in the input cone of the relevant outputs. In the next step, all irrelevant outputs and connected elements are removed as before using Yosys's internal delete and clean passes. They operate on a cell and wire count n_{GFSL} greater than n_{SFGL}. We therefore expect the segmentation time to be higher using GFSL compared to SFGL.

$$n_{\text{GFSL}} > n_{\text{SFGL}} \tag{6.12}$$

Resource consumption of Yosys passes

We analyze the execution times on the AES128-T100 design as provided by Trust-Hub.org.[9][10][11] We chose to use this design since it features a high cell and wire count and two distinct outputs, so we can measure the effects of deleting two outputs. After preparing the design we measure cell and wire counts to

$$c = 345,607 \tag{6.13}$$

$$w = 79,621 \tag{6.14}$$

and with added shadow logic we measure cell and wire counts to

$$c = 1,753,382 \tag{6.15}$$

$$w = 1,274,026. \tag{6.16}$$

Timings were measured for these scenarios:

- AES128-T100 as prepared for the GLIFT pass. Removing the output *out*.
- AES128-T100 as prepared for the GLIFT pass. Removing the output *Capacitance*.
- AES128-T100 with added shadow logic (GLIFT). Removing the output *out*.
- AES128-T100 with added shadow logic (GLIFT). Removing the output *Capacitance*.

Figure 6.2 shows the measured execution times of the passes. The execution time of the delete pass is 0.0127 s for the design with the least cell and wire count (c=39, w=1248) and 0.1931 s for the design with the highest cell and wire count (c=1,753,343, w=1,273,394).

Execution time of the delete pass is nearly constant and therefore, we can give the upper bound $f_{\text{delete}} = \mathcal{O}(1)$. The execution time of the clean pass rises with the wire count, we can therefore provide the lower bound $f_{\text{clean}} \in \mathcal{O}(n)$. With our given set of test designs we can not draw conclusions on the upper bound of the clean pass.

[9] Trust-Hub.org, *Trust-Hub*.
[10] Salmani, Tehranipoor, and Karri, "On design vulnerability analysis and trust benchmarks development".
[11] Shakya et al., "Benchmarking of Hardware Trojans and Maliciously Affected Circuits".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

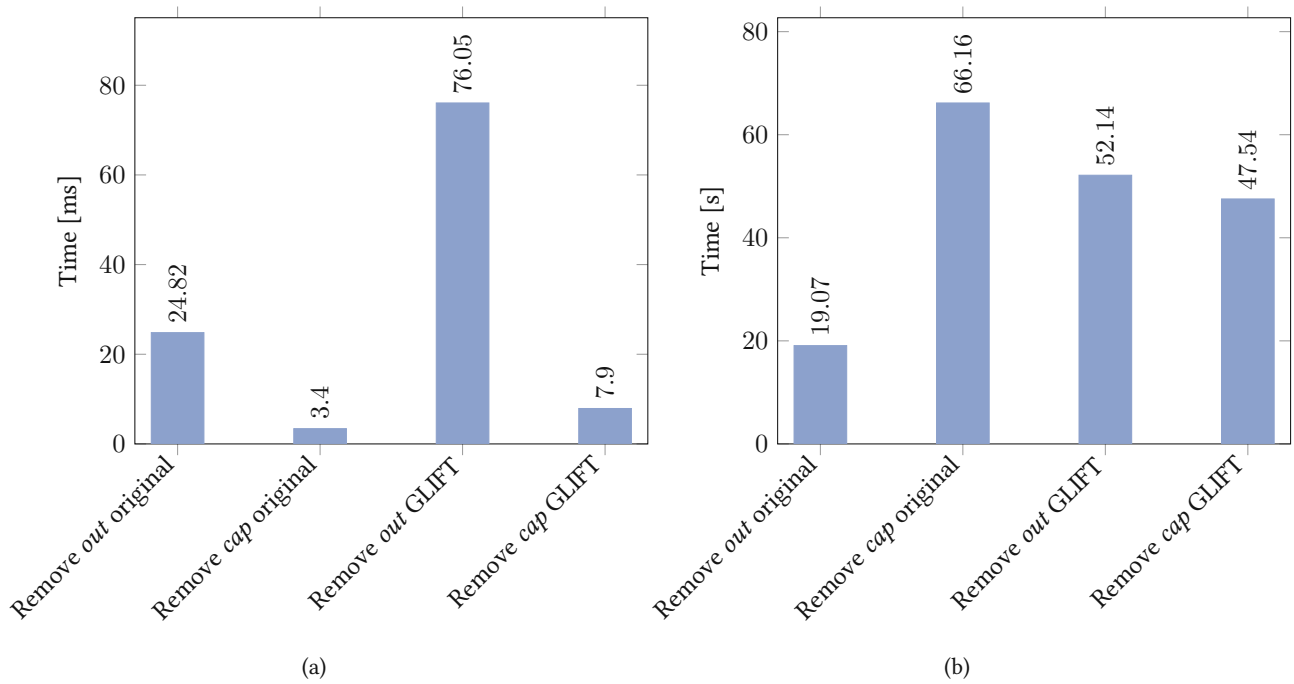


Figure 6.2: Measured execution times of (a) *delete* and (b) *clean* passes.

Figure 6.3 shows the memory consumption of the delete pass has little effect on the memory since it only removes the output wires. The clean pass removes the input cone of this wire and therefore frees more memory. We see that the memory consumed by the synthesis tool is reduced by deleting elements form the design. We achieve this reduction of elements by segmenting the design based on the security domains and the test case. Cell and wire counts for the test designs are shown in Table 6.8. By deleting the output ports we do not remove cells from the design. The number of wire bits is also not influenced since the wire is not actually deleted from the design but actually just marked as a normal wire instead of a port, so it can be deleted by the clean pass. We see that using delete and clean to remove the output *out* removes 2,909,472 (= 99.997 %) cells and 2,926,736 (= 99.35 %) wire bits. It reduces the memory consumption from 9,169 MB to 139 MB which is a reduction of 96.5 %. Removing the Capacitance output results in a reduction of 95 cells (= 0.003 %) and no visible memory reduction since the tool *top* has a resolution of 1 MB. The second test design has shadow logic added and therefore features additional input and output ports for taint values. Also, we performed logic optimization before implementing shadow logic to reduce the memory consumption as adding shadow logic to the design without optimization would yield a memory consumption of $>$ 40 GB. By removing the output `out` we reduce the cell count by 1028 cells (= 0.06 %) and 478,504 wire bits (= 14.8 %) which results in a memory reduction of 0.07 %. Similar, removing the output `Capacitance` removes 39 cells (= 0.002 %) and reduces memory by 1.4 %.

Design	Cells	Wire bits	Memory consumption
Original	2,909,567	2,944,726	9,169 MB
Original - delete out	2,909,567	2,944,726	9,169 MB
Original - clean out	95	17,990	139 MB
Original - delete cap	2,909,567	2,944,726	9,169 MB
Original - clean cap	2,909,472	2,944,407	9,169 MB
GLIFT	1,753,382	3,226,748	3,825 MB
GLIFT - delete out	1,753,382	3,226,748	3,825 MB
GLIFT - clean out	1,752,354	2,748,244	3,772 MB
GLIFT - delete cap	1,753,382	3,226,748	3,825 MB
GLIFT - clean cap	1,753,343	2,750,261	3,771 MB

Table 6.8: Elements in the test designs before and after running Yosys passes used in the segmentation process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

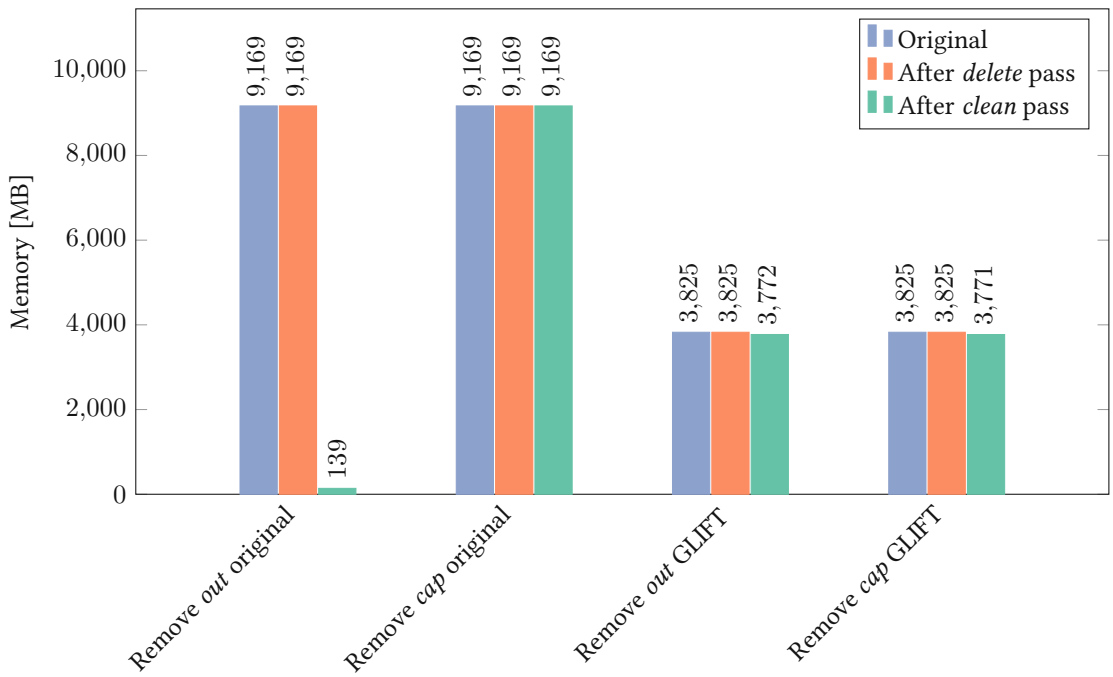


Figure 6.3: Memory consumption of the synthesis tool before and after running delete and clean passes. We measure the memory by deleting outputs from the design AES128-T100 as provided by trust-hub.org . We see that the delete pass does not reduce the memory consumption since it only removes the RTLIL attribute that makes it a port. The resolution of the memory measurement tool is 1 MB, therefore there is no difference in the results after using the delete pass.

Conclusion

From Figure 6.2 we can deduce that the time needed for deleting elements from the design. We expected the complexity of the *delete* pass to be $f_{\text{delete}} = \mathcal{O}(1)$ since it relies on iterating over the design, similar to the *select* pass. The clean pass on the other hand has a higher complexity since it needs to identify and remove cells not used parts of the netlist. We expect this to be at least with a complexity of $f_{\text{clean}} = \mathcal{O}(n)$ which is backed by our experimental results. The design resulting of the segmentation process are equivalent since the same cells and wires get removed compared to the unsegmented design.

We conclude that the execution time of the PGSLIFT script depends on the number of test cases, the size and complexity (the connections between the used cells to form its functionality) of the given hardware design and the structure of the security domain. Figure 6.3 shows how the memory consumption of the synthesis tool depends on the size of the design. While using the *delete* pass only removes the output wire, the clean iterates over the design and removes the whole input cone of the before removed wire. This reduces the memory consumption.

Formal verification

Formal verification analyzes a design by generating a *satisfiability (SAT)* problem and uses so-called SAT solvers to verify against the given properties. Such a SAT problem is proven to be NP complete by the Cook-Levin theorem. A boolean SAT problem can therefore not be solved in polynomial time. Execution times of the verification process therefore relies on the algorithms used by the different SAT solvers. There is a high variability in the performance of these tools. Such solvers are benchmarked on a yearly basis by the *SAT competition.* [12]

[12] http://www.satcompetition.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

SymbiYosys is a front-end to Yosys and is able to execute passes on the design. These so-called *preparation* passes have execution times as shown in Section 6.4. After executing the preparation passes the design's logic function is used to create the SAT problem and the SAT solver is executed. For measuring execution times of the formal verification process we use the *satisfiability modulo theories (SMT)* solver Yices2[13] developed by SRI International. [14]

We provide execution time measurements for each test design in Section 6.2.

Verification options

SymbiYosys configuration uses multiple options that control the verification process. They are provided in the configuration file in the *[options]* section using key value pairs where the value is optional for some options. These options are chosen with the default options in mind for performing formal verification as proposed by the designer of the verification tool. [15]

Options relevant for verifying information flows are:

- *bmc* - Runs a bounded model check to verify properties created using assert statements as shown in Section 5.3.
- *expect* - The expected result of the verification. Influences the return code generated after finishing the verification. Results other than the expected yield a nonzero return code.
- *depth* - Depth for both bmc and prove modes. Determines how many clock cycles the design is verified against the properties. The default value is 20, we reduce it to speed up verification times on large designs such as the AES128-T100. It can be increased using tool options.

To verify our test designs we chose on these options:

Option	Value
bmc	
expect	pass
depth	5

Table 6.9: Chosen options for the verification processes.

As shown in Table 6.9 we run a bounded model check enabled by the *bmc* option. We verify the design again our assertions for five continuous clock cycles which is set by the *depth* option. To automatically check the result of the verification run we need to also control the result code of the script which is set using the *expect pass* option. By expecting the verification to succeed the script returns an error when it does not succeed.

6.5 GLIFT pass - shadow logic creation times

As shown in Section 6.4 the GLIFT pass has linear complexity with the cell count. The number of wires influences the execution time with constant complexity. We measure the execution times for all test designs and provide the mean value of time measurement to create Figure 6.4. A linear curve is fitted to show this linear dependency. With a higher cell and wire count the execution time should rise linear. According to Section 6.4 we expect the time to highly depend

[13] Bruno Dutertre et al. "Yices 2 in SMT-COMP 2019". In: ().
[14] https://yices.csl.sri.com/
[15] https://symbiyosys.readthedocs.io/en/latest/reference.html

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

on the number of cells as the algorithm iterates over all cells in the original design. Since the synthesis tool stores its references to both cells and wires in the same map object and the GLIFT pass also iterates over wires in the design we expect the execution time to also depend on the number of wires in the design. We execute the GLIFT pass five times and measure the mean values. From the mean value we extract the expected execution time of the pass. Table 6.10 shows those mean time values. We see that the execution time rises with a higher cell and wire count. We illustrate these times in Figure 6.4. The big gap between the measurements is due to the difference in cell and wire counts of the designs. We used linear fitting to illustrate the linear dependency. Although four samples are not sufficient to underline this linear dependency, we clearly see the linear trend. Table 6.11 shows the measured values in detail. To better understand the

Design	Cells	Wires	Wire bits	Time
simple-bus	4	8	8	0.001280 s
simple-bus-secure	7	11	11	0.001500 s
AES128-T100	345568	79400	1060497	56.013538 s
RSA-T100	3244	1060	4397	0.515451 s

Table 6.10: GLIFT pass execution times. Cells, wires and wire bits are the initial values before the pass was executed. We clearly see the dependency of the execution time on these counts.

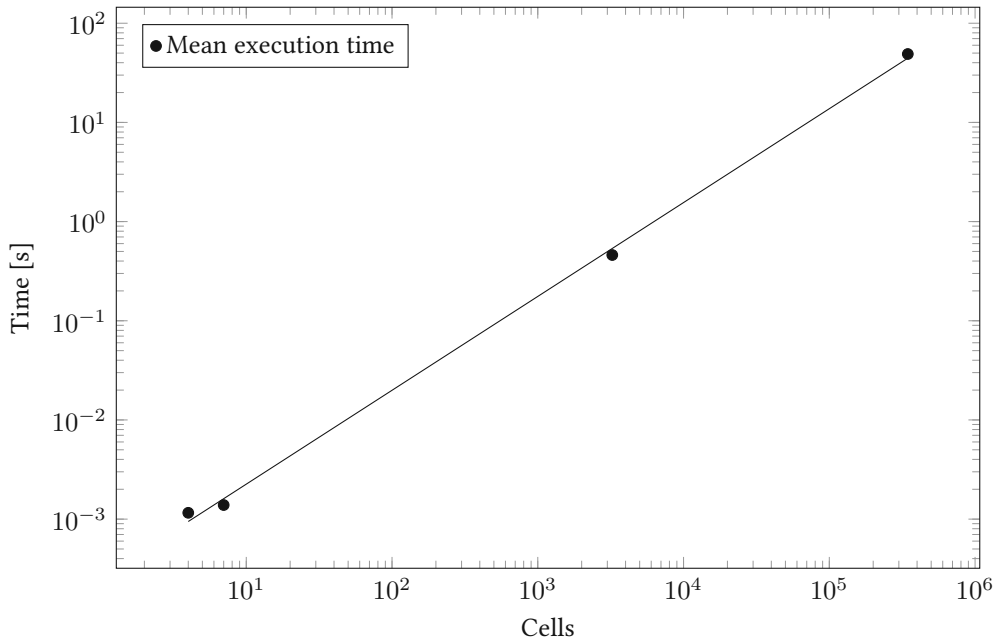


Figure 6.4: Measured execution times of the GLIFT pass for all chosen test designs. Fitting a linear line shows the linear dependency of the execution time on the number of cells in the design. Although, having 4 data points with a diverging distribution in the cell count does not allow drawing conclusions on linear or exponential behavior. Therefore, we perform this test on different test designs to show linear behavior in Figure 6.5.

dependency of the execution time on the elements in the design we tested our GLIFT pass on multiple designs in a less sparse range of cell counts. These designs are used to test the method of Hu et al.[16] and also the implementation of an alternative GLIFT pass that emerged during the writing of this thesis. [17] The design *C7552* and all of its segments have an increased wire count. We randomly segmented this design and so created multiple subsets of it. This design mostly consists of input and output ports which results in a high wire count compared to other designs. We again used linear fitting to create two linear curves that show both the dependency of the execution time on the cell and the wire count. The plot shown in Figure 6.5 shows the results. Table 6.12 shows the results of these measurements.

[16] Hu et al., "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking".
[17] https://github.com/YosysHQ/yosys/pull/20190

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

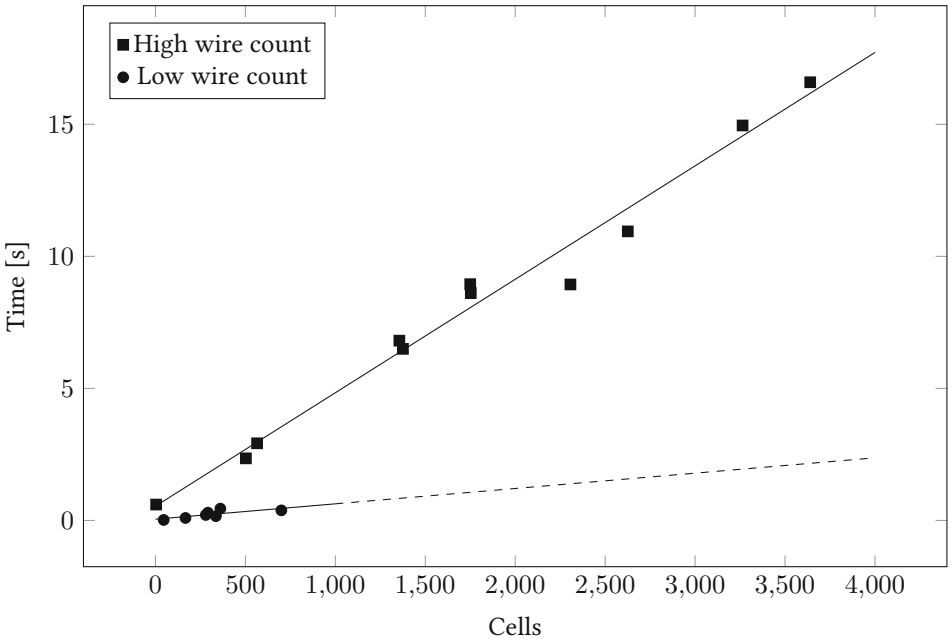


Figure 6.5: Measured execution times of the GLIFT pass for different test designs. A line is fitted for the design C7552 and its derived versions and one for the other test designs with a lower wire count.

Design	Cells	Wires	Wire bits	Time
simple-bus	4	8	8	0.001435 s
simple-bus	4	8	8	0.001168 s
simple-bus	4	8	8	0.001313 s
simple-bus	4	8	8	0.001168 s
simple-bus	4	8	8	0.001314 s
simple-bus-secure	7	11	11	0.001380 s
simple-bus-secure	7	11	11	0.001462 s
simple-bus-secure	7	11	11	0.001384 s
simple-bus-secure	7	11	11	0.001696 s
simple-bus-secure	7	11	11	0.001580 s
AES128-T100	345568	79400	1060497	55.238876 s
AES128-T100	345568	79400	1060497	53.898078 s
AES128-T100	345568	79400	1060497	55.099652 s
AES128-T100	345568	79400	1060497	57.005691 s
AES128-T100	345568	79400	1060497	58.825394 s
RSA-T100	3244	1060	4397	0.500367 s
RSA-T100	3244	1060	4397	0.503608 s
RSA-T100	3244	1060	4397	0.498975 s
RSA-T100	3244	1060	4397	0.510278 s
RSA-T100	3244	1060	4397	0.564029 s

Table 6.11: GLIFT pass execution times details.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Design	Cells	Wires	Wire bits	Time
C7552_lev2_1211.v	4	1211	1211	0.509068 s
t481.v	46	185	185	0.015366 s
ttt2.v	167	668	668	0.095727 s
too_large.v	280	1121	1121	0.200556 s
x1.v	292	1191	1191	0.398671 s
alu2.v	336	1332	1332	0.209221 s
C880.v	361	1459	1459	0.356554 s
C7552_lev2_3308.v	503	3308	3308	2.411875 s
C7552_lev2_4050.v	565	4050	4050	3.143641 s
alu4.v	700	2738	2738	0.560771 s
C7552_lev2_7003.v	1356	7003	7003	6.961974 s
C7552_lev2_6938.v	1376	6938	6938	6.857271 s
C7552_lev2_8316.v	1750	8316	8316	8.794929 s
C7552_lev2_8203.v	1753	8203	8203	9.254923 s
C7552_lev2_10060.v	2307	10060	10060	11.922664 s
C7552_lev2_11128.v	2626	11128	11128	14.293026 s
C7552_lev2_13182.v	3264	13182	13182	18.951673 s
C7552.v	3640	14348	14348	19.889548 s

Table 6.12: GLIFT pass execution times in relation to the cell count of the design. Presented times are the mean value gathered from three repeated measurements.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.6 PGSLIFT - tool execution times

The execution time of the PGSLIFT script is the time needed to analyze a design and create test cases, create the GLIFT shadow logic, segment the design for each test case and create the output files. We compare the implementation times of the different implementation methods presented in Section 5.2. Design preparation is always done beforehand, therefore these times should be equal for all methods but still are relevant since they rely on the complexity of the test designs which we express by the cell and wire count. We test this on all test designs as presented in Section 6.2. Further, tests are done for all three segmentation modes as described in Section 5.2.

Expectations

As shown in Chapter 4, the PGSLIFT script executes the GLIFT pass, extracts test cases, segments the design and then creates output files. Based on the complexity analysis from Section 6.4 we expect the shadow logic creation time to be linear with the cell count.

The complexity of test case analysis is $O(n)$ with n being the number of possible test cases for the given set of input and output domains. We calculate the number of possible test cases using Equation (6.10). For two input and two output domains this yields $n = 60$. As shown in Table 6.7, the times needed for matching regular expressions can be expected By eliminating the test cases that can not be found in the designs input and output security domains we reduce this number. For all provided designs we can reduce to a single test case, that will be verified. Output file creation is done by using Yosys internal passes *write_verilog* which creates a Verilog representation of the design. It's execution time is expected to depend on the cell and wire count with linear complexity.

Results

Times are measured by an automated test script that runs the tool as an external process and parses the outputs created by it. Table 6.13 shows the summary of these measurements. All experiments are executed $n = 5$ times and the mean

Design name	Mode	GLIFT pass	Segmentation	Total
simple-bus	pure	0.0010392 s	0.0000000 s	0.0332172 s
simple-bus	SFGL	0.0010197 s	0.0006770 s	0.0345497 s
simple-bus	GFSL	0.0010756 s	0.0009213 s	0.0348056 s
simple-bus-secure	pure	0.0013440 s	0.0000000 s	0.0401578 s
simple-bus-secure	SFGL	0.0013116 s	0.0007617 s	0.0399203 s
simple-bus-secure	GFSL	0.0013431 s	0.0010456 s	0.0402897 s
AES128-T100	pure	50.6840246 s	0.0000000 s	304.4824618 s
AES128-T100	SFGL	53.1479671 s	35.0358687 s	334.7026849 s
AES128-T100	GFSL	53.4104661 s	56.1186971 s	360.7033699 s
RSA-T100	pure	0.4993984 s	0.0000000 s	3.1208416 s
RSA-T100	SFGL	0.5106142 s	0.2019344 s	3.2404347 s
RSA-T100	GFSL	0.5100762 s	0.4875211 s	3.5099073 s

Table 6.13: Execution times for running the PGSLIFT script for all test designs. The measured times are the mean values gathered from five repeated measurements.

value of the measured times is presented in the summary. Details are shown in Table 6.14. The total execution time is larger than the simple addition of the shadow logic creation time (GLIFT) and the segmentation time. At first, the design is loaded into the synthesis tool which takes up to $\sim$ 50 s for the design AES128-T100. Further, output files are created at the end of the script which can take up to 100 s. We also see that time needed by the PGSLIFT script can be greater when additional segmentation steps are added. This is due to the introduced overhead of segmenting the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

design. We identify the maximal overhead at the test designs AES128-T100 with a maximum of 15.5 % and RSA-T100 with a maximum of 11.1 %. Further, we can conclude that the segmentation method *GFSL* results in a higher execution time than *SFGL* for all test designs. In detail the method GFSL adds 7.77 % execution time for the design AES128-T100 and 8.3 % for the design RSA-T100.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Design	Mode	GLIFT	Segmentation	Total
simple-bus	pure	0.0010150 s	0.0000000 s	0.0316110 s
simple-bus	pure	0.0010140 s	0.0000000 s	0.0328470 s
simple-bus	pure	0.0010520 s	0.0000000 s	0.0332800 s
simple-bus	pure	0.0010770 s	0.0000000 s	0.0331050 s
simple-bus	pure	0.0010380 s	0.0000000 s	0.0352430 s
simple-bus	SFGL	0.0009400 s	0.0012300 s	0.0322750 s
simple-bus	SFGL	0.0008940 s	0.0012320 s	0.0379200 s
simple-bus	SFGL	0.0010160 s	0.0014520 s	0.0370140 s
simple-bus	SFGL	0.0011270 s	0.0015140 s	0.0366240 s
simple-bus	SFGL	0.0010240 s	0.0013420 s	0.0355780 s
simple-bus	GFSL	0.0011980 s	0.0015050 s	0.0370330 s
simple-bus	GFSL	0.0012160 s	0.0014150 s	0.0335590 s
simple-bus	GFSL	0.0010390 s	0.0013010 s	0.0343670 s
simple-bus	GFSL	0.0012570 s	0.0014600 s	0.0358950 s
simple-bus	GFSL	0.0012270 s	0.0013690 s	0.0357330 s
simple-bus-secure	pure	0.0015960 s	0.0000000 s	0.0449880 s
simple-bus-secure	pure	0.0012450 s	0.0000000 s	0.0359790 s
simple-bus-secure	pure	0.0011970 s	0.0000000 s	0.0389700 s
simple-bus-secure	pure	0.0012750 s	0.0000000 s	0.0399320 s
simple-bus-secure	pure	0.0014070 s	0.0000000 s	0.0409200 s
simple-bus-secure	SFGL	0.0013280 s	0.0015340 s	0.0398900 s
simple-bus-secure	SFGL	0.0013430 s	0.0015530 s	0.0368400 s
simple-bus-secure	SFGL	0.0013050 s	0.0016140 s	0.0416180 s
simple-bus-secure	SFGL	0.0013100 s	0.0016000 s	0.0407290 s
simple-bus-secure	SFGL	0.0011100 s	0.0013160 s	0.0393370 s
simple-bus-secure	GFSL	0.0014940 s	0.0017180 s	0.0408080 s
simple-bus-secure	GFSL	0.0015530 s	0.0017370 s	0.0391230 s
simple-bus-secure	GFSL	0.0012930 s	0.0014160 s	0.0406400 s
simple-bus-secure	GFSL	0.0012190 s	0.0015090 s	0.0418410 s
simple-bus-secure	GFSL	0.0014720 s	0.0016870 s	0.0427300 s
AES128-T100	pure	48.7835460 s	0.0000000 s	286.4021880 s
AES128-T100	pure	50.4211280 s	0.0000000 s	306.3060370 s
AES128-T100	pure	51.8783230 s	0.0000000 s	307.9232500 s
AES128-T100	pure	51.1173810 s	0.0000000 s	299.4246740 s
AES128-T100	pure	51.2197450 s	0.0000000 s	322.3561600 s
AES128-T100	SFGL	55.8058790 s	71.8301580 s	376.9328070 s
AES128-T100	SFGL	55.7318690 s	72.1484820 s	370.1064490 s
AES128-T100	SFGL	56.6265180 s	68.6451870 s	359.3927280 s
AES128-T100	SFGL	53.9501000 s	67.7355700 s	356.1474470 s
AES128-T100	SFGL	55.9451820 s	69.9992900 s	362.0351090 s
AES128-T100	GFSL	53.8104080 s	99.6862630 s	423.2453250 s
AES128-T100	GFSL	55.4448430 s	103.8837190 s	430.6525310 s
AES128-T100	GFSL	59.2907420 s	99.6062640 s	417.9632890 s
AES128-T100	GFSL	50.4597960 s	93.9322150 s	395.2572410 s
AES128-T100	GFSL	50.6715320 s	94.3133090 s	396.4053140 s
RSA-T100	pure	0.4995720 s	0.0000000 s	3.2756270 s
RSA-T100	pure	0.5145410 s	0.0000000 s	3.1451740 s
RSA-T100	pure	0.5080700 s	0.0000000 s	3.4805800 s
RSA-T100	pure	0.5148410 s	0.0000000 s	2.9058490 s
RSA-T100	pure	0.4599680 s	0.0000000 s	2.7969780 s
RSA-T100	SFGL	0.5170790 s	0.4291350 s	3.4925550 s
RSA-T100	SFGL	0.4571640 s	0.3718980 s	3.1142270 s
RSA-T100	SFGL	0.4899460 s	0.3891050 s	3.4506300 s
RSA-T100	SFGL	0.6491950 s	0.4370150 s	3.3943760 s
RSA-T100	SFGL	0.4957660 s	0.3921910 s	3.3483510 s
RSA-T100	GFSL	0.5128990 s	1.2382570 s	4.1528780 s
RSA-T100	GFSL	0.4762880 s	0.8227860 s	3.7456850 s
RSA-T100	GFSL	0.5238210 s	1.1883380 s	4.0249310 s
RSA-T100	GFSL	0.4401140 s	0.7503180 s	3.7725520 s
RSA-T100	GFSL	0.5918790 s	1.2937740 s	4.5482170 s

Table 6.14: Detailed execution times of running the PGSLIFT script.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.7 Formal verification times

We tested the verification times for all designs presented in Section 6.2 after running the PGSLIFT script as shown in Section 6.6. To get a statistical relevant result we repeat the measurement $n = 5$ times. We further test the verification time for both the unsegmented and the segmented design to conclude on the impact of segmentation. All verification time tests are done using an automated test script that performs the tests, measures the results and generates the tables we provide.

As shown in Section 6.4, we expect the shadow logic creation time to increase with increasing cell and wire count. We expect this time to also be shorter when the segmentation method *SFGL* or *GFSL* is used where the design's cell and wire counts are reduced before the GLIFT pass is executed. Similarly, we expect the time needed for the segmentation to increase with an increased cell and wire count since it uses internal passes provided by the synthesis tool. The additional overhead created by the segmentation adds to the shadow logic creation time, therefore we expect the total execution time to increase when segmentation is used in some cases. Table 6.15 shows the median of measured times

Design name	Verification 1	Verification 2	Verification total	Script total
simple-bus	0.250 s	0.416 s	0.666 s	0.686 s
simple-bus segmented	0.242 s	0.361 s	0.603 s	0.625 s
simple-bus-secure	0.359 s	0 s	0.359 s	0.374 s
simple-bus-secure segmented	0.347 s	0 s	0.347 s	0.360 s
AES128-T100	794.274 s	1082.197 s	1878.469 s	1878.484 s
AES128-T100 segmented	743.268 s	1033.271 s	1776.537 s	1776.749 s
RSA-T100	6.606 s	10.814 s	17.420 s	17.439 s
RSA-T100 segmented	6.347 s	10.422 s	16.769 s	16.786 s

Table 6.15: Execution times for running the verification tool *verify.py* for all test designs. The measured times are the mean values gathered from five repeated measurements.

for all designs. Designs with a found information leak require more than one execution of the verification since the verification process ends immediately when the assertions break. The second verification time is for the case that there is no information flow detected in the design. Raw values are provided for each design the consumed times as shown in Table 6.16. The total execution time of the script includes file handling and result creation.

simple-bus

Verification result in two executions where at the first an information flow is detected. The second verification is execution succeeds and ends the script. This applies to both the segmented and the unsegmented version of this design. The first execution of the verification terminates faster since an error is found, whereas the second execution executes to the maximal depth set by the configuration file. We see that the segmented design terminates faster by 9.3 % in the first and 7.6 % in the second execution. As shown before, the segmented design features 12.7 % fewer cells than the unsegmented design.

simple-bus-secure

Since this design has the additional IFT control module that avoids the "leak state" of the select inputs (0,0) Similar to the results from the unsecure design, we see that the verification time can be reduced by 5.6 % when verifying the segmented design. We can further compare the verification times to the design without the IFT control module by comparing the last verification times from each design. This reduces by 15.7 % at the unsegmented and 3.9 % on the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

segmented versions.

AES128-T100

The identification of the information leak results in a verification time of 794 s which can be reduced by 6.8 % to 743 s when verifying the segmented design. When verifying for repeated clock cycles the verification time can be reduced from 1082 s to 1033 s.

RSA-T100

Our method identifies an information flow in 6.606 s. With the absence of this information flow verification takes 10.8 s since the assertions are tested for multiple clock cycles. Using the segmented version of the design reduces the first verification attempt by 4.1 % to 6.347 s, the second attempt by 4.5 % to 10.422 s.

Design name	Verification 1	Verification 2	Verification total	Script total
simple-bus	0.255 s	0.37 s	0.626 s	0.646 s
simple-bus	0.262 s	0.399 s	0.662 s	0.682 s
simple-bus	0.234 s	0.385 s	0.619 s	0.636 s
simple-bus	0.236 s	0.563 s	0.798 s	0.818 s
simple-bus	0.262 s	0.361 s	0.623 s	0.646 s
simple-bus segmented	0.232 s	0.362 s	0.593 s	0.616 s
simple-bus segmented	0.232 s	0.357 s	0.589 s	0.612 s
simple-bus segmented	0.254 s	0.353 s	0.607 s	0.631 s
simple-bus segmented	0.237 s	0.357 s	0.595 s	0.614 s
simple-bus segmented	0.255 s	0.375 s	0.63 s	0.651 s
simple-bus-secure	0.342 s	0 s	0.342 s	0.356 s
simple-bus-secure	0.337 s	0 s	0.337 s	0.351 s
simple-bus-secure	0.356 s	0 s	0.356 s	0.374 s
simple-bus-secure	0.382 s	0 s	0.382 s	0.394 s
simple-bus-secure	0.380 s	0 s	0.38 s	0.396 s
simple-bus-secure segmented	0.358 s	0 s	0.358 s	0.376 s
simple-bus-secure segmented	0.353 s	0 s	0.353 s	0.367 s
simple-bus-secure segmented	0.340 s	0 s	0.34 s	0.35 s
simple-bus-secure segmented	0.331 s	0 s	0.331 s	0.341 s
simple-bus-secure segmented	0.353 s	0 s	0.353 s	0.368 s
AES128-T100	793.990 s	1083.17 s	1887.161 s	1887.182 s
AES128-T100	795.002 s	1076.293 s	1871.295 s	1871.312 s
AES128-T100	794.009 s	1086.351 s	1880.36 s	1880.371 s
AES128-T100	794.216 s	1084.465 s	1878.681 s	1878.696 s
AES128-T100	794.153 s	1080.694 s	1874.847 s	1874.862 s
AES128-T100 segmented	743.965 s	1034.203 s	1778.168 s	1778.184 s
AES128-T100 segmented	744.060 s	1038.933 s	1782.993 s	1783.933 s
AES128-T100 segmented	742.922 s	1031.889 s	1774.811 s	1774.826 s
AES128-T100 segmented	744.341 s	1030.481 s	1774.822 s	1774.85 s
AES128-T100 segmented	741.050 s	1030.848 s	1771.899 s	1771.951 s
RSA-T100	6.580 s	11.055 s	17.635 s	17.655 s
RSA-T100	6.785 s	10.562 s	17.347 s	17.359 s
RSA-T100	6.540 s	10.821 s	17.361 s	17.374 s
RSA-T100	6.488 s	10.748 s	17.236 s	17.254 s
RSA-T100	6.639 s	10.885 s	17.523 s	17.537 s
RSA-T100 segmented	6.094 s	10.71 s	16.803 s	16.816 s
RSA-T100 segmented	6.224 s	10.541 s	16.765 s	16.784 s
RSA-T100 segmented	6.519 s	10.379 s	16.898 s	16.915 s
RSA-T100 segmented	6.468 s	10.249 s	16.717 s	16.732 s
RSA-T100 segmented	6.432 s	10.23 s	16.662 s	16.683 s

Table 6.16: Detailed execution times of running the verification script on our test designs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 7

Discussion and Conclusions

7.1 Discussion

Creating a simple and flexible security policy using PGSL

We demonstrate that we can create security policies using a hierarchy of security domains expressed as a PGSL pattern. This pattern reassembles a security policy that can be used to identify information leaks in hardware designs. This leak is identified by implementing GLIFT and using formal verification. Compared to the state-of-the-art method SecVerilog, this security policy is more simple and human-readable since it consists of interconnections of security domain names and is expressed in a single statement. We demonstrate the creation of the PGSL pattern using an exemplary design called simple-bus, shown in Section 4.5. An unintended information flow that leaks a secret key to an output can be identified using our method. By adding an IFT control module we control the access to the bus and therefore remove the unintended information flow. Our method does not detect an information leak in this modified design since the leak is avoided by our modification. The formal verification process finishes by verifying the assertions for five continued clock cycles. To benchmark our method we use the exemplary designs used to introduce the methodology as well as the test designs AES-T100 and RSA-T100, gathered from trust-hub.org.[1][2] The latter both represent a *hardware Trojan (HT)* that leaks the key used in the cryptographic process to the output. We create security domains so, that the key is in a security domain of a higher level than that of the output. By using our method on those designs we identify an information leak like in the design simple-bus. We provide reports created by our tool chain in Listing 4.11, Listing 4.12, Listing 6.4 and Listing 6.5. They show the verification results and the state of inputs that lead to the information leak being detected.

Overhead of creating shadow logic and partitioning the design

By analyzing the algorithm used to create shadow logic based on a given hardware design we show that its execution time is of linear complexity $f_{\text{glift}} = \mathcal{O}(n)$ with n being the number of cells in the design. Our experimental results confirm this complexity analysis and show that the number of cells in the design influences the execution time linearly as shown in Figure 6.4. We tested this linear dependency for designs with a maximum of 3640 cells, therefore we can

[1] Salmani, Tehranipoor, and Karri, "On design vulnerability analysis and trust benchmarks development".
[2] Shakya et al., "Benchmarking of Hardware Trojans and Maliciously Affected Circuits".

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

prove this dependency until this cell count.

We show how the number of cells and wires are increased by adding shadow logic in Section 6.2 and summarize the measured values in Table 7.1. The cell count increases between the factor 4.6 and 8.5 depending on the test design. In wire bits we see an increase between the factor 3.1 and 5.4, also depending on the test design.

Design name	Cells original	Cells GLIFT	Wire bits original	Wire bits GLIFT
simple-bus	4	23	8	32
simple-bus-secure	12	55	19	74
AES128-T100	345,568	1,753,304	1,060,497	3,235,778
RSA-T100	3,244	21,194	4,397	23,594

Table 7.1: Summary of ell and wire overhead when adding shadow logic to a design.

Our method analyzes the designs based on the security policy and removes parts of the design that do not influence the verification results. We call this process *segmentation*. By segmenting a hardware design and therefore reducing the cell and wire count we try to reduce the execution time of shadow logic creation which is also confirmed by our experimental results as can be seen in Table 6.13. The total time is the execution time of the Python script, which we present in Section 5.2. It includes pattern parsing, test case extraction, segmentation, shadow logic creation and output generation. From the measurements we conclude that the overall execution time of adding shadow logic can not be reduced using segmentation. Depending on the design and the test case, the amount of cells purged from the design vary as shown in Section 6.2. For our designs there is so little change in design size that the actual shadow logic creation time can not be reduced significantly. Since segmentation of the design also adds execution time, the resulting total execution time also rises by a maximum of 15.5 % for the test design AES128-T100 that has the highest cell and wire count. For segmenting the design, two methods were implemented and analyzed. Calculating their complexity results in the same upper bounds $f_{\text{segmentation}} = \mathcal{O}(n)$. Results showed that the segmentation method *segmentation first glift later (SFGL)*, as introduced in Section 5.2, has a reduced execution time and only increases the execution time by 8.3 % compared to *glift first segmentation later (SFGL)* for the test design RSA-T100. Overall, segmentation using SFGL adds 30.2 s to the execution time for the test design AES128-T100, 120 ms for RSA-T100 and 1.3 ms for the design simple-bus. We see a reduction of total execution time of 0.2 ms for the design simple-bus-secure.

Memory consumption of the synthesis software is also influenced by the number of cells and wires in the test design. We show that removing 5,836,208 of 5,854,293 cells and wires (= 99.67 %) from a design reduces the memory consumption of the synthesis tool by 96.5 %. Removing 476,526 of 4,980,130 cells and wires (= 0.06 %) of cells and wires in a different test shows a reduction of memory consumption by 0.07 %.

Execution time of formal verification

The methods used by the verification process solve a known NP-complete problem. From that we assume that the verification time can be reduced by segmenting the hardware design since it is an exponential function of the numbers of elements in the design $t_{\text{verification}} = f(n_{\text{cells}}, n_{\text{wires}})$. We confirm this by conducting experiments that execute the verification on the test designs in both a segmented and an unsegmented state. Our experimental results show that the amount of reduction of verification time depends on the number of remaining cells and wires. Table 6.15 shows the measured times for running the verification on our test designs in both unsegmented and segmented variants. We see that the execution times of the verification process differs between the designs. For the test design *simple-bus* this took

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

in average 250 ms for the unsegmented and 242 ms for the segmented design variants that is a reduction by 3.5 %. The second verification attempt where is no information flow in the design took 416 ms and 361 ms which is a reduction of 13.2 %. Verifying the modified design takes 359 ms for the unsegmented and 347 ms for the segmented variant which is a reduction of 3.3 %. In the design AES128-T100 we identify the information leak in 794 s and 743 s which is a reduction of 6.4 %. Similar we see a reduction of the verification time for the RSA-T100 design from 6.606 s to 6.347 s which is a reduction of 3.9 %. For the simple-bus designs we see reductions of 3.2 % and 3.3 %. We conclude that segmentation has a measurable effect on the verification time. Also, the number of elements that can be removed using segmentation vary between designs.

Comparing the time added by the segmentation $t_{\text{segmentation}}$ and the time reduction of formal verification $t_{\text{reduction}}$ shows that the time reduction is greater than the overhead.

$$t_{\text{reduction}} = |\Delta t_{\text{verification}}| \tag{7.1}$$

$$t_{\text{segmentation}} < t_{\text{reduction}} \tag{7.2}$$

The total execution time t_{total} of our method can therefore be reduced using segmentation. Overall we see a reduction in t_{total} of our tool by 21 s (= 1.9 %) on the test design AES-T100. For some designs this reduction could be insignificant whereas when verifying a design with multiple times the elements this reduction can be important. Even though the advantages are not very obvious from the usage of segmentation when comparing the elements removes, the results from the verification time comparison are showing that it can have a measurable effect.

Comparing to previous work

We compare our method to the state-of-the-art method SecVerilog.[3] They implement security policies by creating the so-called security lattice. It is implemented in SMT-LIB syntax and their default lattice, consisting of 4 security domains, requires 6 SMT statements to be created. Each security domain is defined as a label and every transition using a function. We show that by using a security policy expressed as a one line PGSL pattern our method requires fewer changes of source code and fewer added lines. Our security policies can therefore be changed and modified with less effort for the verification engineer. Also, our policy is more simple and easy to understand since it is an interconnection of security domain names without additional statements. With more sophisticated attacks we identify the need for security policies that are more simple and easy to understand. Using PGSL we achieve this pursued level of simplicity.

As shown in Chapter 4 our method introduces the security policy in a single statement which can be compared to a single line of code. Further, our method introduces information on the security domain by adding Verilog attributes to all input and output ports of the design. Since we use Verilog attributes, there is no need to remove them from the design as they have no influence on synthesis tools as shown by Sutherland.[4] Table 7.2 shows a comparison of properties of SecVerilog

Property	SecVerilog	PGSLIFT
Policy syntax	SMT-LIB	PGSL
Policy lines	9	1
Security labels	non-standard	Verilog attributes
Changed HDL lines	5	4

Table 7.2: Comparing SecVerilog to PGSLIFT.

[3] Zhang et al., "A hardware design language for timing-sensitive information-flow security".
[4] Sutherland, *Verilog® HDL*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

and PGSLIFT. We see that the creation of a security policy requires fewer lines of code and therefore adaptions of the policy are also possible with less effort.

Limitations

Using our method we treat the design as a black box for applying security policies via a PGSL pattern. The design itself needs to be available as a white box to create shadow logic based on its gate level netlist. The presented method therefore is a gray box approach. Using the formal verification tool SymbiYosys allows us to test these properties by applying assertions and assumption to the designs inputs and outputs. We can not easily provide detailed information on where in the design the actual information flow occurred.

Not all hardware designs can be assigned to a security model with strict domains since there may be some potion of intended information flow to outputs of lower domains. An encryption process for example encrypts information and distributes it to an output in a security domain of a lower level. Future works could adapt a PGSL pattern similar to that used by our method to describe the flow of information at designs internals. To do so, information needs to be added to the HDL sources.

Other state-of-the-art approaches like SecVerilog[5] use SAT solvers to create a list of input states that lead to information flowing from the designs inputs to its outputs. Our chosen verification method creates a report and terminates as soon as a violation of the assertions is found. The report provides information about the input state that lead to the assertion. With this information the design can be examined and fixed.

Future work

This work provides a proof of concept that creating security policies using PGSL is possible, and they can be used to identify information leaks violating these policies. From the conclusions and limitations of this thesis we extract possible improvements to increase usage of our work and counteract the limitations.

In-design domain changes

Similar to the state-of-the-art method SecVerilog[6] we could introduce functions that describe the change of security domains for elements in the design. Similar functions could be expressed as PGSL patterns similar to the already used pattern to introduce a domain hierarchy. Information about where a domain change happens in the design could be added using Verilog attributes similar to those we do for adding security domain information to ports of the design. This way the added statements would not influence the normal synthesis of the design and would therefore not need to be removed. Further, we could exchange the formal verification tool with common SAT solvers to create a list of found information flows and make a decision on intended or not intended by evaluating the result with respect to the defined domain change functions.

Quantitative properties

By introducing information about the quantity of an information flow the verification engineer could for example model how many output bits can have a taint. This will allow the modelling of more complex information flows. In future

[5] Zhang et al., "A hardware design language for timing-sensitive information-flow security".
[6] Ibid.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

work this is possible by using the versatile syntax of PGSL and adding information on how many information bits are allowed to flow against the domain hierarchy. Using more complex, but still automatically generated, SystemVerilog assertions allows to verify these modified security policies.

Abstraction levels

The presented method uses GLIFT which analyzes the design at a gate level stage. By implementing *information flow tracking (IFT)* at higher levels of abstraction, such as *register transfer level (RTL)*, the overhead that comes with creating shadow logic at gate level could be reduced as shown by Ardeshiricham et al.[7] The design could be examined as a RTL netlist. State-of-the-art method *register transfer level information flow tracking (RTLIFT)*[8] shows that this reduces the amount of false positives and also lowers the execution time of the verification process. We also identify the possibility of increasing the abstraction level even more and perform information flow tracking on the behavioral level to create a methodology similar to IFT methods in software development, such as Jif.[9]

Tool upgrades

The tools are written in Python which allows for wide-ranging modifications. For example, a graphical interface could be added to the tools using *tkinter* or similar libraries. The design could be presented in an abstract way and the assignment of security domains to ports would be possible without deeper knowledge of the HDL or our method. This could boost the usage of our method and the used formal verification tools and ease the process for verification engineers and those who are yet to be.

7.2 Conclusions

This work shows that a flexible security policy, created using a PGSL pattern, can be used to verify a hardware design for information flow and detect information leaks. It can further be used to segment the design in order to reduce verification time. Such a policy can be expressed by a single PGSL pattern making it simple to understand and easy to use. In Chapter 4 (Methodology) we show the process of identifying security domains, putting them in a hierarchical order and create a PGSL pattern that is the security policy. Our method uses this policy to add shadow logic to the design and then segment and verify the design using state-of-the-art hardware verification tools. Using standard Verilog attributes we modify the design in a way, so it can be used as-is in a normal synthesis process and is not influenced by our additions. Experiments done on our test designs showed that the method is capable of identifying information flows that violate our security policies. An exemplary verification report is shown in Section 4.5 (PGSLIFT example). We further test our method on benchmark designs AES128-T100 and RSA-T100 gathered from trust-hub.org.[10][11] These findings answers research question 1.

By segmenting the design based on the PGSL pattern we create subsets of it which can be verified independently. Formal verification techniques rely on solving SAT problems which are proven to be NP-complete by the Cook-Levin theorem.[12] We experimentally measure verification times for designs that were segmented and unsegmented. In Section 6.6

[7] Ardeshiricham et al., "Register transfer level information flow tracking for provably secure hardware design".
[8] Ibid.
[9] CS Cornell, *Jif - Java Information Flow Tracking*.
[10] Salmani, Tehranipoor, and Karri, "On design vulnerability analysis and trust benchmarks development".
[11] Shakya et al., "Benchmarking of Hardware Trojans and Maliciously Affected Circuits".
[12] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

(PGSLIFT - tool execution times) we showed that the verification time is reduced when verifying such a segmented design by a maximum of 15 % depending on the amount of reduced design elements. The number of removable elements depends on the hardware design and can be higher in other designs. Further, we show that resource consumption in terms of memory of the verification process is also reduced when verifying the segmented designs. Again, this reduction depends on the number of removable elements in the design.

Creating subsets of a design also adds additional time and resource overhead created while processing the design. We analyze this overhead by performing a complexity analysis in Section 6.4 and then perform experiments to both support our analysis and claims in Section 6.5 and Section 6.6. The complexity analysis shows a linear complexity of our tools. We were able to confirm this for designs with up to 3640 cells with experimental results. The findings above answers research question 2a. We measure a minimum time reduction of the verification process of 3 % and a maximum of 6.4 % which answer research question 2b. Depending on the DUT the number of removable elements by segmentation could be greater and the impact therefore more significant. The time reduction in the verification process is greater than the overhead added by segmentation. The total execution time of our method can therefore be reduced using segmentation. We measure an overall time reduction of 1.9 % for the test design AES128-T100.

We conclude that we were able to proof the usability of PGSLIFT in modern hardware security applications. Our proposed security policies are simple to use and easy to understand which eases the process for the verification engineer and reduces the probability of errors done when creating the policy. The open syntax of PGSL enables future work on our method by adding additional security attributes. We were able to reduce to total execution time of our tool by removing elements from the design that do not affect the verification result. We see this work as a basis for future development and improvement.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Bibliography

[1] Armaiti Ardeshiricham. "Verification and Synthesis of Information Flow Secure Hardware Designs". PhD Thesis. UC San Diego, 2020.

[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. "Register transfer level information flow tracking for provably secure hardware design". In: Mar. 2017, pp. 1691–1696.

[3] Elliott D. Bell. "Looking Back at the Bell-La Padula Model". In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005.

[4] Bell, D Elliott and LaPadula, Leonard J. *Secure computer systems: Mathematical foundations*. Tech. rep. MITRE CORP BEDFORD MA, 1973.

[5] Swarup Bhunia, Michael S. Hsiao, Mainak Banga, and Seetharam Narasimhan. "Hardware Trojan Attacks: Threat Analysis and Countermeasures". In: *Proceedings of the IEEE* 102.8 (Aug. 2014), pp. 1229–1247.

[6] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. en. Tech. rep. Apr. 1977. Chap. Technical Reports. (Visited on 03/04/2021).

[7] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 1971.

[8] CS Cornell. *Jif - Java Information Flow Tracking*. 2016.

[9] Howard S. Dakoff. "The clipper chip proposal: Deciphering the unfounded fears that are wrongfully derailing its implementation". In: *J. marshall L. rev.* 29 (1995), p. 475.

[10] Bruno Dutertre, Dejan Jovanovic, Ian A Mason, and Stéphane Graham-Lengrand. "Yices 2 in SMT-COMP 2019". In: ().

[11] J. Ferrigno and M. Hlavac. "When AES blinks: Introducing optical side channel". In: *Information Security, IET* 2 (Oct. 2008), pp. 94–98.

[12] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. "ECDSA key extraction from mobile devices via nonintrusive physical side channels". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1626–1638.

[13] Daniel Genkin, Adi Shamir, and Eran Tromer. "Acoustic Cryptanalysis". en. In: *Journal of Cryptology* 30.2 (Apr. 2017), pp. 392–443. (Visited on 05/19/2021).

[14] Inc. Gentoo Foundation. *SELinux/Information flow control*. [Online; state: March 4 2021]. 2014.

[15] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy*. ISSN: 1540-7993. Apr. 1982, pp. 11–11.

[16] Aarti Gupta. "Formal hardware verification methods: A survey". In: *Formal Methods in System Design* 1.2-3 (Oct. 1992), pp. 151–238.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[17] Seokhie Hong. *Side Channel Attacks.* MDPI-Multidisciplinary Digital Publishing Institute, 2019. ISBN: 9783039210015.

[18] W. Hu, J. Oberg, J. Barrientos, D. Mu, and R. Kastner. "Expanding Gate Level Information Flow Tracking for Multilevel Security". In: *IEEE Embedded Systems Letters* 5.2 (June 2013), pp. 25–28.

[19] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. "Theoretical Fundamentals of Gate Level Information Flow Tracking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.8 (Aug. 2011), pp. 1128–1140.

[20] Wei Hu, Ardeshiricham Armaiti, and Ryan Kastner. "Hardware Information Flow Tracking". In: (2020).

[21] Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. "Imprecise Security: Quality and Complexity Tradeoffs for Hardware Information Flow Tracking". In: *Proceedings of the 35th International Conference on Computer-Aided Design.* ICCAD '16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450344661.

[22] k0resh. *Timing attacks - part 1.* en. Dec. 2013. (Visited on 05/27/2021).

[23] Kadri Kaska, Henrik Beckvard, and Tomas Minarik. "Huawei, 5G and China as a security threat". In: *NATO Cooperative Cyber Defence Center for Excellence (CCDCOE)* 28 (2019).

[24] Christoph Kern and Mark R. Greenstreet. "Formal verification in hardware design". In: *ACM Transactions on Design Automation of Electronic Systems* 4.2 (Apr. 1999), pp. 123–193.

[25] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP).* ISSN: 2375-1207. May 2019, pp. 1–19.

[26] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Annual international cryptology conference.* Springer, 1999, pp. 388–397.

[27] François Koeune and François-Xavier Standaert. "A Tutorial on Physical Security and Side-Channel Attacks". en. In: ed. by Alessandro Aldini, Roberto Gorrieri, and Fabio Martinelli. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 78–108. ISBN: 9783540319368. (Visited on 03/11/2021).

[28] Christian Krieg. "Pattern-Based Hardware Trojan Characterization for Design Security Assessment". PhD thesis. Gusshausstrasse 27–29 / 384, 1040 Wien: Vienna University of Technology (TU Wien), Feb. 2019.

[29] Christian Krieg, Clifford Wolf, and Axel Jantsch. "Malicious LUT: A Stealthy FPGA Trojan Injected and Triggered by the Design Flow". In: *Proceedings of the 35th International Conference on Computer-Aided Design.* ICCAD '16. Austin, Texas: ACM, 2016, 43:1–43:8. ISBN: 978-1-4503-4466-1.

[30] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. "Information flow control for standard OS abstractions". In: *ACM SIGOPS Operating Systems Review* 41.6 (Oct. 2007), pp. 321–334.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". en. In: 2018, pp. 973–990. ISBN: 9781939133045. (Visited on 03/11/2021).

[32] Ashok B. Mehta. *SystemVerilog Assertions and Functional Coverage.* Springer, 2020.

[33] Prabhat Mishra, Swarup Bhunia, and Mohammad Tehranipoor. *Hardware IP security and trust.* eng. Cham: Springer, 2017. ISBN: 3319490249.

[34] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

[35] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*. 1999, pp. 228–241.

[36] Andrew C. Myers and Barbara Liskov. "A decentralized model for information flow control". In: *ACM SIGOPS Operating Systems Review* 31.5 (Dec. 1997), pp. 129–142.

[37] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor. "Hardware trojan detection through information flow security verification". In: *2017 IEEE International Test Conference (ITC)*. Oct. 2017, pp. 1–10.

[38] A. J. Neumann, N. Statland, and R. D. Webb. "Post Processing Audit Tools and Techniques". In: *Proceedings of the NBS Workshop, Miami Beach, Florida, USA*. 1977, pp. 36–341.

[39] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. "Theoretical analysis of gate level information flow tracking". In: *Design Automation Conference*. ISSN: 0738-100X. June 2010, pp. 244–247.

[40] Kyle Pullicino. "Jif: Language-based Information-flow Security in Java". In: *CoRR* abs/1412.8639 (2014).

[41] H. Salmani, M. Tehranipoor, and R. Karri. "On design vulnerability analysis and trust benchmarks development". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. Oct. 2013, pp. 471–474.

[42] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. "Benchmarking of Hardware Trojans and Maliciously Affected Circuits". In: *Journal of Hardware and Systems Security* 1.1 (Mar. 2017), pp. 85–102.

[43] Stuart Sutherland. *Verilog® HDL*. 2001. ISBN: 1-930368-03-8.

[44] Vaibbhav Taraate. *SystemVerilog for Hardware Description: RTL Design and Verification*. Springer Nature, 2020.

[45] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. "Complete Information Flow Tracking from the Gates Up". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 109–120. ISBN: 978-1-60558-406-5.

[46] Trust-Hub.org. *Trust-Hub*.

[47] Benedikt Tutzer, Christian Krieg, Clifford Wolf, and Axel Jantsch. "Python Wraps Yosys for Rapid Open-Source EDA Application Development". In: 2019.

[48] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. "Labels and event processes in the Asbestos operating system". In: *ACM Transactions on Computer Systems* 25.4 (Dec. 2007), p. 11.

[49] Vivek Venugopalan and Cameron D. Patterson. "Surveying the Hardware Trojan Threat Landscape for the Internet-of-Things". en. In: *Journal of Hardware and Systems Security* 2.2 (June 2018), pp. 131–141. (Visited on 03/11/2021).

[50] Claire Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.

[51] Clifford Wolf. *SymbiYosys*. https://github.com/YosysHQ/SymbiYosys.

[52] Steve Zdancewic. "Challenges for information-flow security". In: Citeseer, 2004, p. 6.

[53] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making information flow explicit in HiStar". In: *Communications of the ACM* 54.11 (Nov. 2011), pp. 93–101.

[54] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. "A hardware design language for timing-sensitive information-flow security". In: *ACM SIGARCH Computer Architecture News* 43.1 (2015), pp. 503–516.

[55] Douglas E. Zongker. "Chicken Chicken Chicken : Chicken Chicken". In: https://isotropic.org/papers/chicken.pdf. 2002.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Appendix A

Appendix

A.1 Simple-bus

```
module simple_bus (i0, i1 , o0, o1, sa, sb
  );
  (*sec_domain="KEY"*) input i0;
  (*sec_domain="DATA"*) input i1;

  (*sec_domain="UART"*) output o0;
  (*sec_domain="SPI"*) output o1;

  (*sec_domain="none"*)input sa;
  (*sec_domain="none"*)input sb;

//Instantiate the top module
top top (i0, i1, o0, o1, sa, sb);

endmodule
```

Listing A.1: Wrapper that adds security domains to the top module.

```
module top (i0, i1 , o0, o1, sa, sb);
  input i0;
  input i1;
  input sa;
  input sb;
  output o0;
  output o1;
  wire o_mux;

mux_2x1 mux (i0, i1, sa, o_mux);
demux_1x2 demux(o_mux, sb, o0, o1);

endmodule
```

Listing A.2: Top module connecting MUX and DEMUX.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
module mux_2x1 (i_0, i_1, s, o);
input i_0;
input i_1;
input s;
output o;

//use assign to select between input1 and input2
assign o = s ? i_0 : i_1;

endmodule
```

Listing A.3: MUX module.

```
module demux_1x2 (a, s, o_0, o_1);
input a;
input s;
output o_0;
output o_1;

assign o_0 = ~s & a;
assign o_1 = s & a;

endmodule
```

Listing A.4: DEMUX module.

A.2 Simple-bus-secure

To secure the design we modify the top module to include the control module and connect it to the select inputs of MUX and DEMUX modules.

```
module top (i0, i1 , o0, o1, sa, sb);
  input i0;
  input i1;
  input sa;
  input sb;
  wire sa_save;
  wire sb_save;
  output o0;
  output o1;
  wire o_mux;

control ctrl (sa, sb, sa_save, sb_save);
mux_2x1 mux (i0, i1, sa_save, o_mux);
demux_1x2 demux(o_mux, sb_save, o0, o1);

endmodule
```

Listing A.5: Secure top module that introduces the control module.

```
module control(
input sA,
input sB,
output sA_save,
output sB_save
);

assign sA_save = sA ^ sB;
assign sB_save = sA | (~sB);
endmodule
```

Listing A.6: Control module that controls the select inputs of MUX and DEMUX.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

A.3 Tools

Implementation hurdles

During the implementation we were able to identify some problems in Pyosys. We were able to solve them using alternative implementations.

Pyosys provides access to Yosys's internal data structure by wrapping up C++ structures with Python structures.

- Wrapping C++ structures is done in a universal attempt and therefore special cases are not always included.
- Some structures and methods are not fully exposed through the wrapper.
- The combination of using Pyosys through libyosys leads to a lack of information on available objects and methods in available IDEs. Therefore, the developer using Pyosys is required to study Yosys source-files to find out what information of the netlist could be exposed and then determine what is actually exposed using a trial-and-error approach
- Some structures can not be referenced directly due to technology-differences of C++ and Python (compiling vs interpreting, ...) and therefore some elements are copied when being accessed. This leads to unavoidable memory and computation overhead that effects the performance on bigger designs. We were able to bypass this problem by creating lists of object and creating lookup dictionaries.
- The reference to *libyosys* differs whether the Python module is by Yosys or from a system Python interpreter. When run from the system interpreter, it expects the file at the "dist-packages" directory inside the module "Pyosys". Yosys has its own reference to libyosys and therefore expects it to be directly available. We resolve this problem by importing the library depending on the *__name__* variable as shown in Listing A.7.

```
1 if __name__ == "__main__":
2     from pyosys import libyosys as ys
3 else:
4     import libyosys as ys
```

Listing A.7: We import libyosys from different locations depending on the __name__ variable.

- **Dealing with multi-bit wires** Wires can represent more than one bit. A wire is therefore represented by a *SigSpec* object which includes a list of *SigBit* object. These object represent the actual bit that is connected to a logic gate. When analyzing a taint wire we therefore need to analyze the corresponding SigSpec to get a list of its SigBits.
- **Constants at cell inputs** Gate inputs can either be attached to a bit of a wire found in our lookup dictionaries or a constant driver. For example, this is done when comparing a signal to a constant value. Constructs like that reassemble an *if* clause.

  Using a single *AND* gate that has one input set to constant 1, its output is only 1 if the remaining input is also 1. In Yosys these constants are represented with a driver being connected to a logic gate. Under normal circumstances all taint-signals are propagated from input ports to gate. When dealing with constants we do not have a propagated taint signal and therefore need to assign a taint signal ourselves. We assume that there is no information on

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the constant signal and therefore assign the taint input the value 0. This does not affect the taint information propagation. When analyzing the input of a gate we therefore detect if the input is a constant signal or a bit that carries taint information and decide where the taint input connects to.

- **Map between original and tainted bits** RTLIL does not provide unique identifiers to access SigBit objects. In order to store the relation between wire bits of the original design to those of their taint equivalents we need a way of storing this relationship.

  Therefore, when traversing the original design and generating shadow logic, we need to access the corresponding taint bit for every bit in the original design. Wire objects do support a unique representation, implemented by the IdString class. As a consequence we generate a lookup dictionary that maps each SigBit found in the original designs wires to those of the cloned taint wires. Then we are able to access single taint bits by accessing the tainted wire at the offset of the original bit inside the original wire.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct - Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Vienna, Austria Oktober 2021

Philipp Wassertheurer, BSc

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub