Security Verification of Compiler Optimizations: An Information Flow Perspective

Thesis submitted to the
Indian Institute of Technology Guwahati
for the award of the degree

of

Doctor of Philosophy

in

Computer Science and Engineering

Submitted by
Priyanka Panigrahi

Under the guidance of
Dr. Chandan Karfa

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

November, 2023

Copyright © Priyanka Panigrahi 2023. All Rights Reserved.

Dedicated to

My Loving Parents

for their unwavering support and encouragement throughout my academic journey. Their sacrifices and belief in my abilities have been the driving force behind my academic success.

Declaration

I certify that:

- The work contained in this thesis is original and has been done by myself and under the general supervision of my supervisor.
- The work reported herein has not been submitted to any other Institute for any degree or diploma.
- Whenever I have used materials (concepts, ideas, text, expressions, data, graphs, diagrams, theoretical analysis, results, etc.) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references. Elaborate sentences used verbatim from published work have been clearly identified and quoted.
- I also affirm that no part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.
- I am fully aware that my thesis supervisor is not in a position to check for any possible instance of plagiarism within this submitted work.

Date: November 24, 2023
Place: Guwahati

(Priyanka Panigrahi)

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Chandan Karfa, for his unwavering guidance, invaluable insights, and continuous support throughout the entirety of my Ph.D. journey. His expertise and guidance have been instrumental in shaping both the direction and quality of this research. I extend my sincere acknowledgment to the members of my dissertation committee, Prof. Jatin Kumar Deka, Prof. Arijit Sur, and Dr. John Jose, for their constructive feedback, scholarly contributions, and dedicated involvement in the development of this work. Each member has played a crucial role in refining the conceptual framework and methodology of the study. I am indebted to my past dissertation committee member, Dr. Arnab Sarkar for providing valuable feedback during the initial days of my research. I express my heartfelt gratitude to my collaborator Prof. Prabhat Mishra - I am incredibly grateful and privileged to have had this opportunity to work with him. I am grateful to Qualcomm India for providing financial support for my research. The assistance from both my mentors Bernard Nongpoh and Nilo Redini from Qualcomm have allowed me to focus on my academic pursuits and has contributed significantly to the successful completion of this Ph.D.

My heartfelt thanks go to my fellow researchers at AVS lab, my lab seniors Ramanuj Bhaiya, Surajit Sir, and Mohammed, and my juniors Praveen, Akash, and especially Nilotpola whose intellectual and collaborative spirit have created a stimulating academic environment. The exchange of ideas and experiences has enriched my research and made this academic endeavor both challenging and fulfilling. I got the opportunity to work with Sahitya, Abhik, Vignesh, Birjit, and Karthik. Thank you for sharing insights and technical discussions countless times which helped me carry out my research work.

To my friends, Debabrata Sir, Swagat Sir, Shakeel, Siva, Gyanendro, Prasen, Sumita, Aswathy, Roushni, Kashmiri, and Pallabi, thank you for your encouragement and motivational support. I want to thank my seniors and juniors, especially, Aparajita Mam, Anasua Mam, Khushboo Mam, Shrestha, Sheel, Ujjwal Da, Bhale Sir, Arijit Sir, Deepika Mam, Divya Mam, Manjari Mam, Alakesh, Nilotpal, Maithilee, Nidhi, Vanshali, Karnish, Komal, Saurav, Manoj, Swati, and many others. To my dearest friends Pratibha, Tasrin, Emte, Tinka, Khyati, Lolly (Nilotpola), and Udangshree, I am grateful for the fun-filled, enjoyable moments we spent together, and the late-night unlimited conversations at the hostel during the ups and downs of this Ph.D. journey. Your consistent emotional support has been a constant source of strength for me.

I want to thank my late paternal grandparents (Maa n Bapa), my late maternal grandparent (Aja), my maternal grandparent (Aai), my uncles and aunts (Bada Mausi n Mausa, Sana Mausi n Mausa, Bada Mamu n Mai, Sana Mamu (late) n Mai), my cousin brothers and sisters (Papun bhai, Chiku, Sagar, Pati, Pritam, Prem, Sikha, and Deepa), and the new family members (Rosy Bhauja n Rimi) for the WhatsApp group video calls and best wishes throughout this journey.

Finally, I want to express my gratitude to my pillars of strength - my parents (Bou n Baba), my elder sister (Apa), and my brother-in-law (Jiju) for your unconditional love, encouragement, patience, and support during this crucial journey. Your belief in my potential has been a source of motivation. I am extending a special gratitude to my mother (Bou) for your constant care and daily updates about the family. From the early stages of my research to the final stages of completing this journey, her encouragement has been my guiding light. To my nephew (Vishnu) and my niece (Urvi), thank you for your laughter and boundless energy, which provided a welcome respite from the challenges of research. Your presence has added a special and meaningful dimension to this journey, and I am grateful for the shared moments of joy. I am expressing my heartfelt thanks to the special person in my life - my husband-to-be (Saroj) for being a constant source of support and love in this journey. In the moments of self-doubt, his presence and belief in my abilities have provided comfort and strength. Our shared anticipation of the future has made every accomplishment more meaningful, and I am fortunate to have his supportive companionship.

November 24, 2023 **Priyanka Panigrahi**

Certificate

This is to certify that this thesis entitled, **"Security Verification of Compiler Optimizations: An Information Flow Perspective"**, being submitted by **Priyanka Panigrahi**, to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a bonafide work carried out by her under my supervision and guidance. The thesis, in my opinion, is worthy of consideration for the award of the degree of Doctor of Philosophy in accordance with the regulation of the institute. To the best of my knowledge, it has not been submitted elsewhere for the award of the degree.

Date: November 24, 2023
Place: IIT Guwahati

....

Dr.Chandan Karfa
Associate Professor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Abstract

Modern compilers like GCC, LLVM apply various optimizations on the source program to improve the performance of the target code for execution time, code size, resource usage, memory usage, etc. One of its critical requirements is to generate a functional equivalent target code. A target code generated after application of compiler optimization may be functionally equivalent to the source program but it may not be as secure as the source program (i.e., relatively secure). Therefore, it is essential to ensure that the optimized code does not introduce any security vulnerability during the optimization phase. This thesis aims to verify the relative security between the source and optimized programs, irrespective of the optimizations applied by a compiler. Specifically, the information flow is considered as the security property in a program in this thesis. To achieve relative security, we first aim to quantify the information leakage in a program using static taint analysis. Then, we propose a bisimulation method for translation validation of information leakage for relative security verification between a source and an optimized program. The next work explores how a model checker can be utilized to quantify the information leakage in a program. The model-checking-based security analysis method can further be applied to translation validation of information leakage for relative security verification between the source and optimized programs. With our notion of relative security, we have shown that the register allocation step in a compiler is not secure in the presence of spilling. We then propose a secure register allocation approach for the LLVM compiler framework. Finally, this thesis aims to protect these registers from information leakage, specifically from scan-based attacks.

Keywords- *Compiler*, *Security*, *Information Flow*, *Information Leakage*, *Register Allocation*, *LLVM*, *Taint Analysis*, *Model Checking*, *Formal Verification*, *CBMC*, *Scan Chain*, *Side Channel*, *Register Transfer Level.*

Contents

List of Figures

List of Tables

List of Symbols

M_0	Source FSMD
M_1	Optimized FSMD
q_{00}	Reset state of M_0
q_{10}	Reset state of M_1
τ_0,τ_1	Traces in FSMDs M_0 and M_1
α, β	Paths in FSMD
α^s	Start state of path α
α^f	Final state of path α
R_α	Condition of execution of the path α
S_α	Data transformations of the path α
γ_α	Leak vector of the path α with no initial leak
γ^{α^s}	Initial leak at α^s
$\gamma_\alpha^{\alpha^s}$	Leak vector of the path α with initial leak at α^s
$\gamma_\alpha\|_{(c,h_j)}$	Corresponding bit b_j for j^{th} high input in c of leak vector γ_α
$\gamma_\alpha\|_{(v_i,h_j)}$	Corresponding bit b_{ij} for j^{th} high input in i^{th} program variable v (i.e., n_i) of leak vector γ_α

γ_{loop}	Overall leak of the loop
$leaky_h$	Number of leaky high inputs
$leaky_{uv}$	Number of unique leaky variables
$leaky_v$	Number of leaky variables
$\beta \simeq_S \alpha$	β is securely equivalent to α
$\beta \simeq_R \alpha$	β is relatively secure to α
$\beta \simeq_C \alpha$	β is conditionally secure to α
$M_1 \simeq_S M_0$	M_1 is securely equivalent to M_0
$M_1 \simeq_R M_0$	M_1 is relatively secure to M_0

1 Introduction

Compilers play a pivotal role in software development in modern computing systems. A compiler converts a source program into a semantically equivalent target program. The source programs are the human-readable high-level programs, and the target programs are the assembly or machine-executable programs. A compiler performs a series of crucial tasks as shown in Fig. 1.1 to achieve this [17]. It involves two parts: analysis and synthesis. In Fig. 1.1, the first three steps are the analysis phase, and the last four steps are the synthesis phase of the compiler.

1.1 Compilation Steps

A compiler generates the machine code from a source code using the following steps [98]:

1. *Lexical Analysis:* It is the initial phase of the compiler, and it scans the source code and breaks it down into smaller units called tokens. It identifies keywords, operators, and identifiers. This helps in simplifying the subsequent processing.

2. *Syntax Analysis:* It checks for the program structure and grammar rules of the source language. This step produces an abstract syntax tree (AST) representing the program

structure. In an AST, each interior node represents an operation, and the children of the interior nodes represent the arguments of the operation.

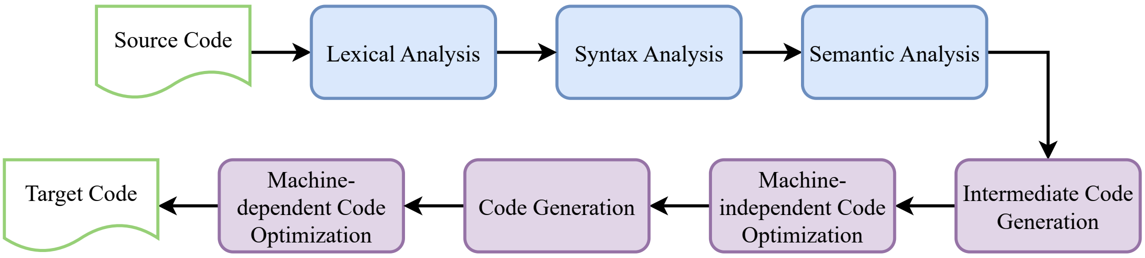

Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Machine-independent Code Optimization
Code Generation
Machine-dependent Code Optimization
Target Code

Fig. 1.1: *Phases of a Compiler*

3. *Semantic Analysis:* This phase uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It checks for type errors, variable declarations, and function calls, ensuring the program adheres to the semantics of the language.

4. *Intermediate Code Generation:* The compiler generates one or more intermediate representations in the process of translating a source code into machine executable code. Syntax trees used in the syntax and semantic analysis phases of the compiler is one form of intermediate representation. Three-address code is another form of intermediate code generated by compilers.

5. *Machine-independent Code Optimization:* This phase of the compiler attempts to improve the intermediate code by enhancing the efficiency of the generated code concerning faster execution, shorter code, less power, etc. It also aims to reduce the memory usage while preserving the functional correctness.

6. *Code Generation:* It takes an optimized intermediate representation of the source code and maps it into the target language. This phase produces the machine code or assembly language that the machine can execute.

7. *Machine-dependent Code Optimization:* This phase is often referred to as low-level code optimization. It generates code for a specific target machine or hardware architecture to maximize performance and efficiency. Different hardware platforms have unique features, instruction sets, and memory hierarchies that can be leveraged for

optimization. In this phase, register allocation is performed, where registers and/or memory locations are selected for each of the variables in the program.

The code optimization is a primary and crucial step in the compilation process. Machine-independent optimizations focus on improving the code performance without any assumptions about the target hardware. These optimizations generate more portable code which can run on various architectures. However, machine-dependent optimizations are specific to a particular target hardware. Generic optimizations like common subexpression elimination, dead code elimination, constant folding, etc., are examples of machine-independent optimizations. Machine-dependent optimizations include selecting the most appropriate instructions from the target architecture instruction set, assigning the registers to program variables judiciously, minimizing cache misses and exploiting data locality by effective memory access patterns, choosing the addressing modes that align with the target architecture capabilities, and many more. We discuss the code optimization phase of the compiler below.

1.1.1 Compiler Optimization

Modern compilers, like GCC, CLANG, and LLVM, have various optimization levels, like, -O0, -O1, -O2, and -O3, that control the aggressiveness of optimizations. The -O0 level disables almost all optimizations. This level focuses on the compilation time of the generated code. The -O1, -O2, and -O3, levels apply basic, moderate, and high optimizations, respectively. In -O1, basic optimizations like common subexpression elimination, constant folding, and dead code elimination are applied without a significant increase in compilation time. The level -O2 applies function inlining, loop unrolling, loop fusion, etc., in addition to the optimizations applied in -O1. This level focuses on improving the performance of the generated code. The optimization level -O3 applies more aggressive optimizations like loop interchange, loop distribution, instruction scheduling, etc., in addition to the optimizations applied in -O2. Some of the optimizations are applied in almost all the levels, like copy propagation and code motion, to improve the code quality and performance. The aggressiveness of these optimizations depends on the chosen optimization level and the capabilities of the compiler. There are other levels of optimizations, such as -Os for size optimization, -Ofast for faster execution, and -Og optimization for debugging. Compilers allow to specify custom optimization flags or enable specific optimizations individually. The compilers -O1 and -Os optimization levels often include machine-independent optimizations. The other levels mostly include machine-dependent optimizations.

This optimizations must be implemented in a precise manner to ensure the correctness of the compiler [83, 104]. Any deviation from the source code's intended behavior by the compiler can introduce bugs, vulnerabilities, and any undefined consequences in the compiled program. Now, we discuss the various techniques adopted for ensuring the correctness and security of a compiler. We focus on verifying whether a functionally correct compiler can lead to security vulnerabilities in the generated target code.

1.2 Correctness of Compiler

From a given source code, compilers aim to generate functionally equivalent target code that would lead to faster execution in the target architecture. Functionally equivalence means that for all possible inputs, both source and target codes terminate and produce the same output values. A compiler is correct if it produces functionally equivalent target code from all possible input programs. Since compilation is a complex process involving various steps, as mentioned above, a compiler may have implementation bugs. In fact, many bugs are reported on popular compilers like LLVM or GCC [5, 6]. Consequently, a compiler may produce functionally incorrect code. Hence, checking the correctness of a compiler is an important step. Here, we present a limited list of approaches for verifying the functional correctness of a compiler.

1. *Testing* verifies the outputs of the target code with the golden outputs for each test input of the program. A failure in testing implies the presence of errors in the program. It includes unit testing, integration testing, and regression testing [40].

2. *Theorem Proving* is the process of verifying the behavior of a system mathematically. It requires formal methods and mathematical proofs [84–86].

3. *Model Checking* intends to verify whether the implementation meets the given specification which is provided as temporal properties [46].

4. *Translation Validation* proves that each translation of the compiler is functionally correct. In other words, it checks equivalence between the source code and the target code generated by the compiler [69, 81, 102, 111].

5. *Static Analysis* includes identifying the errors at the compile time of the programs. It helps to identify the issues related to data flow and control flow.

6. *Dynamic Analysis* includes identifying the errors during the run time of the programs.

7. *Fuzz Testing* involves subjecting the compiler to a large number of randomly generated inputs to uncover unexpected behaviors in the optimized code [41, 50].

8. *Symbolic Execution* allows the exploration of different execution paths of a program symbolically and further helps to analyze the effects of optimizations on program behavior systematically [22, 77].

Informally, it may seem that a functionally correct compiler should also preserve security properties, but this is not so. Correctness and security turn out to be two distinct issues. In this thesis, we assume the optimized code is functionally correct to the source code, and we are focusing on analyzing the security of the generated optimized code. We now discuss the security of the compiler in the next Section.

1.3 Security of Compiler

An optimized code must be checked for various security properties to preserve source-level security [107], like confidentiality, integrity, deterministic memory allocation, finite memory size, undefined behavior, information flow, well-typedness, well-bracketed control flow, continuation manipulation for declassification leak, network-based threats, etc. Security is an important concern in many embedded and cyber-physical systems. Therefore, it is important to ensure that the generated code preserves the security properties of the source program. There exist numerous works [76, 111] on verifying the functional correctness of the generated code. However, there is not much study on the security vulnerability of the generated code. Embedded compilers should be aware of security implications when applying optimizations. As identified in the literature [56], the compiler optimizations are one of the sources of security flaws introduced in the generated code. Some optimizations might inadvertently weaken the source-level security to improve the performance of the target code. Security-aware compilation ensures that the compiler does not compromise security-critical parts of the code during optimization.

We have discussed various security properties in Section 2.1 of the next chapter. Information flow security property has many application domains, like Android, WebAssembly, JavaScript, etc. The compiler may introduce new information flow or modify the existing information flow in a program due to some optimizations it applies. In our observation, the impact of compiler optimizations on the security of a program can be measured best by analyzing the information flow property. Therefore, in this thesis, we verify a program's information flow security property. Note that an optimized program may have more leaks due to violating other security properties.

1.3.1 Information Flow Security Property

Information flow in a program concerned about the secure information flow, i.e., from high-security or private data to low-security or public variables. The private data, like passwords and cryptographic keys, are inputs of the program or read from tamper-proof memory, whereas the public data are available through the variables and outputs of the program. The user must mention the sensitivity of each input. Information flow is of two types: explicit and implicit. The code of Fig. 1.2(a) presents an example of explicit/direct information flow where the value of a high-security input h is stored in a low-security variable x. A more subtle form of information flow emerges in the context of indirect information propagation, as illustrated in Figure 1.2(b). In this scenario, we consider a situation where the attacker lacks direct access to the memory but can still identify whether a value is 0 or not by monitoring the output of the low-security variable x. This phenomenon is referred to as implicit/indirect information flow because the mere presence of a high-security value in the control condition of a branching construct influences the outcome of that branch. A target programming language lacking safeguards against information flow is susceptible to such leaks. Information flow, thus, violates the confidential property as well.

1.3.2 Information Flow Tracking in a Program

Taint analysis [31,42,43] is a widely applied technique to track information flow in a program through taint flows. Tainted variables generally depend on user input directly or indirectly. Generally, if a variable is untainted, it contains non-sensitive values. In other words, there is no information flow from the sensitive inputs to the untainted variables. So, there is no leak through the untainted variable. It has a wide range of practical applications such

```
void func(int h)
{
  int x = h;
  ...
}
```

(a)

```
void func(int h)
{ int x;
  if (h == 0)
   x = 0;
  else
   x = 1;
    ...
}
```

(b)

Fig. 1.2: *Examples of Information Flow: a) Explicit Information Flow, b) Implicit Information Flow*

as software vulnerability detection [74, 92], privacy leak detection [33, 71, 101], malware detection [60] for android platform [128]. It generally has two categories: (i) Static taint analysis [27, 37, 54, 64, 91] detects taint flows across all possible program paths at compile time, and (ii) Dynamic taint analysis [47, 75, 120] detects taint flows at run time in a single execution trace.

The major problem with the traditional taint analysis method is under-tainting and over-tainting. It under-approximates the leak by ignoring the implicit information flow [19, 66] and over-approximates the leak by considering all variables defined inside a tainted conditional block (i.e., the condition of the block is tainted) as tainted [37, 91]. Under-tainting leads to false negative scenarios and over-tainting causes false positives. Moreover, these works either ignore loops in the program or consider a single path inside a loop during implicit leak identification. In this thesis, we plan to use the static taint analysis technique to overcome the issue of both under-tainting and over-tainting.

1.4 Motivation and Objectives

This thesis primarily targets security verification of compiler optimizations with respect to information flow in a program. In this Section, we discuss the motivations along with objectives behind the research works carried out in this thesis.

1.4.1 Security Analysis of Compiler Optimization Techniques

A functionally correct compiler optimization may not be always secure [56]. Consider dead store elimination (DSE), which removes the dead code from the program. Let us analyze the relative security between the programs before and after DSE in Fig. 1.3. The compiler applies DSE to remove the store zero operation in Fig. 1.3(a), considering it as a dead code, and generates the target code in Fig. 1.3(b). DSE is functionally correct, but it does not preserve the security of the source program since the sensitive information remains in variable x till the end of execution of the program. This gives an attacker more opportunity to get sensitive information. Therefore, it is essential to identify and ensure that compiler optimizations preserve the source level security [27, 54, 56, 108].

```
void foo()
{
      x = password();
      ...//use x;
      x = 0; //dead store
      ...//rest of the code
}
```

(a)

```
void foo′()
{
      x = password();
      ...//use x;

      ...//rest of the code
}
```

(b)

Fig. 1.3: *An Example of Dead Store Elimination (DSE): a) Source code, b) After DSE*

The security of common compiler optimizations and/or transformations like DSE, static single assignment (SSA), etc., have been analyzed, and a secure version of the same has been reported in [52, 53]. Let us now consider an important and essential optimization step in a compiler, i.e., register allocation (RA), in which the variables of a program are mapped to possibly fewer physical registers. Two or more variables can be mapped to a single register if their lifetimes do not overlap [39, 112]. If the number of registers is not sufficient to map the variables of a program, the compiler finds a set of suitable variables to split. During the RA process, a compiler may choose to split a single variable into multiple registers if the live ranges of those registers do not overlap. Live range splitting [48] helps to assign the variables into a specific number of registers without using memory (spilling). Typically, splitting is applied before spilling. However, for moderately sized programs, it is often not feasible to allocate all variables to registers, which requires the use of spilling. Spilling [38] involves storing the value of a variable in memory when there are no available registers. To use a spilled variable, a load from memory is required before its use, and a store to memory is necessary after the variable is defined. Although register allocation is a mandatory step in any compilation flow, its security analysis has not been explored yet. With the above motivation on security of compiler optimization and background of RA, we formulate the following objective:

Objective 1: A compiler like GCC and LLVM converts a source program into a target assembly after register allocation. Our first objective in this thesis is to analyze the security issues in register allocation. We try to answer an important question: is register allocation secure with respect to preserving information leakage? If not, propose a secure register allocation approach for a real-time compiler.

1.4.2 Relative Security Verification

In secure compilation, the compilation steps are shown not to introduce any security vulnerabilities. Secure compilation involves implementing security measures within the compiler itself. Certifying the security of a compiler, however, is a hard problem [27,54]. One way of securing compiler optimization is to develop a secure version of each compiler optimization, such as secure dead store elimination [52], secure static single assignment [53], etc. These secure versions of the optimizations are restrictive in nature. In certain situations, employing these secure optimizations on source code might be unnecessary, as many of these optimizations are optional and depend upon the specific characteristics and behavior of the source code. In contrast, register allocation is a mandatory optimization performed by the compiler. Thus, we attempt to secure the register allocation in Objective 1. However, ensuring the security of each compiler optimization is a challenging problem as modern compiler applies hundreds of optimizations. A realistic approach is the translation validation of the security of compiler optimization in which the overall security of the source and the optimized program will be compared to check if the optimization phase has introduced any information leakage. Translation validation is a novel approach that could offer an alternative to the security verification of compiler optimization. This approach will formally check if the target code is securely equivalent to the source program after each execution of the compiler. Although there exist works for checking the functional correctness of compilers using translation validation [14,69,77,81,87,122], none of them is applicable to the security verification of compiler optimizations. Moreover, in the context of translation validation of information leakage of compiler optimizations for security correctness, the important question arises: "How to define the relative security of the optimized program with respect to the source program?" In the same context, another significant question arises: "How do we measure the security of both source and transformed programs to check the relative security between them?" Here, the relative security implies that the transformed program is as secure as the source program. Note that, relative security does not ensure the security of the source program, that there is no information leakage in the source program. The optimized program can be relatively secure to the source one, and the source program can still be leaky. To check the relative security between the source and optimized programs, we have the following objectives in the thesis.

Objective 2: Our second objective is to quantify the information leakage in a program concerning information flow. Our objective is to develop a precise taint analysis method that primarily overcomes the problem of over-approximation in the prior works. Our goal is to use this quantification method to verify the relative security between a source program and its optimized program.

Objective 3: The third objective of this thesis is to develop a translation validation approach to verify the relative security between a source program and its optimized program based on the quantification method proposed to achieve our second objective. The idea here is to perform a bi-simulation on both the source and optimized program in a path-based manner [1].

1.4.3 Information Leakage Detection with Model Checker

The C Bounded Model Checker (CBMC) [2] is an open-source tool to check for the specified functional correctness and security properties in the form of assertions. To handle loops, it explores all execution paths within user-specified bounds. The CBMC can handle C programs that involve arrays, functions, pointers, and all other constructs in C. It is commonly used in safety-critical and security-critical software development, where correctness and reliability are paramount. It generates a counter-example if it detects a violation of a specific property. The counter-example is a concrete execution trace that can demonstrate the information flow property violation for our purpose. The basic flow of a model checker is shown in Fig. 1.4.

[1] A path is an execution flow between two program points. We define the path formally in Section 4.3.1

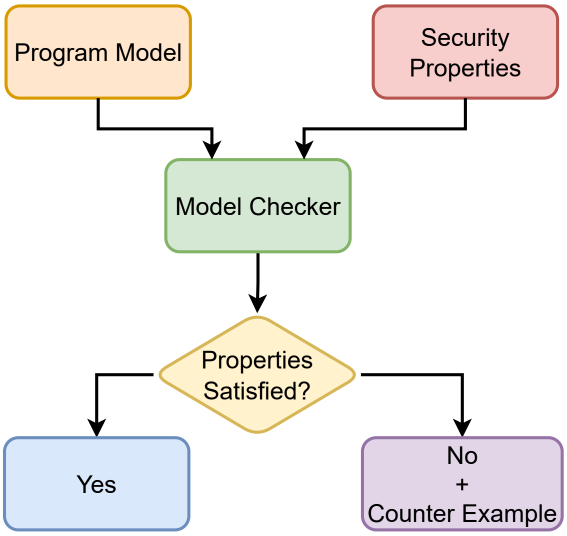


Fig. 1.4: *Model Checker*

Objective 4: Our next objective is to formally model the information flow property of a program as user-specified assertions and utilize the potential of the CBMC to verify them. Specifically, we aim to develop a practical tool to measure the information leak for a compiler like LLVM. We then extend the CBMC based method to verify the relative security between the source and the generated optimized code of a compiler.

1.4.4 Securing Registers from Information Leakage with High-level Synthesis

We discussed so far the motivation towards a securing compilation intended for execution on general-purpose processors. However, the generated machine code (typically assembly language) by these compilers is generally not highly portable across different hardware architectures, as it is optimized for a specific target architecture. Also, the execution time of a program in general-purpose processors is generally high. The High-level synthesis (HLS) [95] is used to convert high-level software descriptions, typically written in C or C++ into a register transfer level design (RTL) in hardware description languages (HDLs) such as VHDL or Verilog. Such RTL is executed in a specialized platform like a Field-Programmable Gate Array (FPGA) or an integrated circuit (IC) can be obtained from it for Application-Specific Integrated Circuit (ASIC) targets. The output of HLS is effectively a hardware accelerator (HA) for the input program, which is much faster as compared to its execution in general-purpose processors.

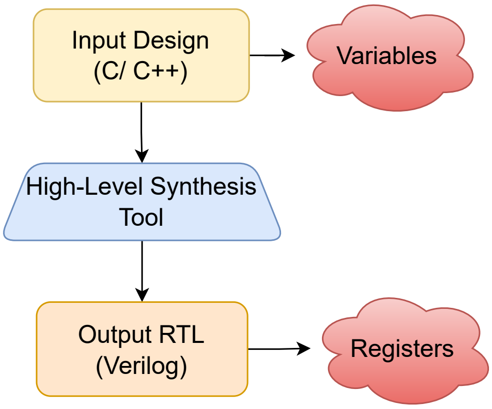


Fig. 1.5: *C to RTL generation through HLS*

In objectives 2 and 4, we find the total information leakage in a program due to the information flow. The variables in a source program are mapped to registers in the hardware during HLS. Thus, sensitive information stored in the variables is now held by the registers in the corresponding RTL generated by HLS. The basic flow of RTL generation is shown in Fig. 1.5. We now formulate the following objective to analyze the information leakage through the registers in the HLS generated RTLs.

Objective 5: The final objective of this thesis is to protect the sensitive registers from leaking information at the RTL design generated by HLS. To achieve this, we aim to propagate the information flow identified in the source program to the RTLs through HLS. Our next objective is to corrupt the content of sensitive registers in the RTL without changing the input-output functionality of the design. This will protect the registers from leakage if the attacker has access to the registers in hardware.

1.5 Contributions

This thesis developed different approaches to quantify, verify, and mitigate the information leakage of compiler optimizations. The entire research carried out in this thesis is comprised of five contributory chapters. Each of these chapters is targeted to analyze the security of a program from the perspective of information flow. An overview of the contribution of the thesis is presented in Fig. 1.6. The contributions of the thesis are summarized below.

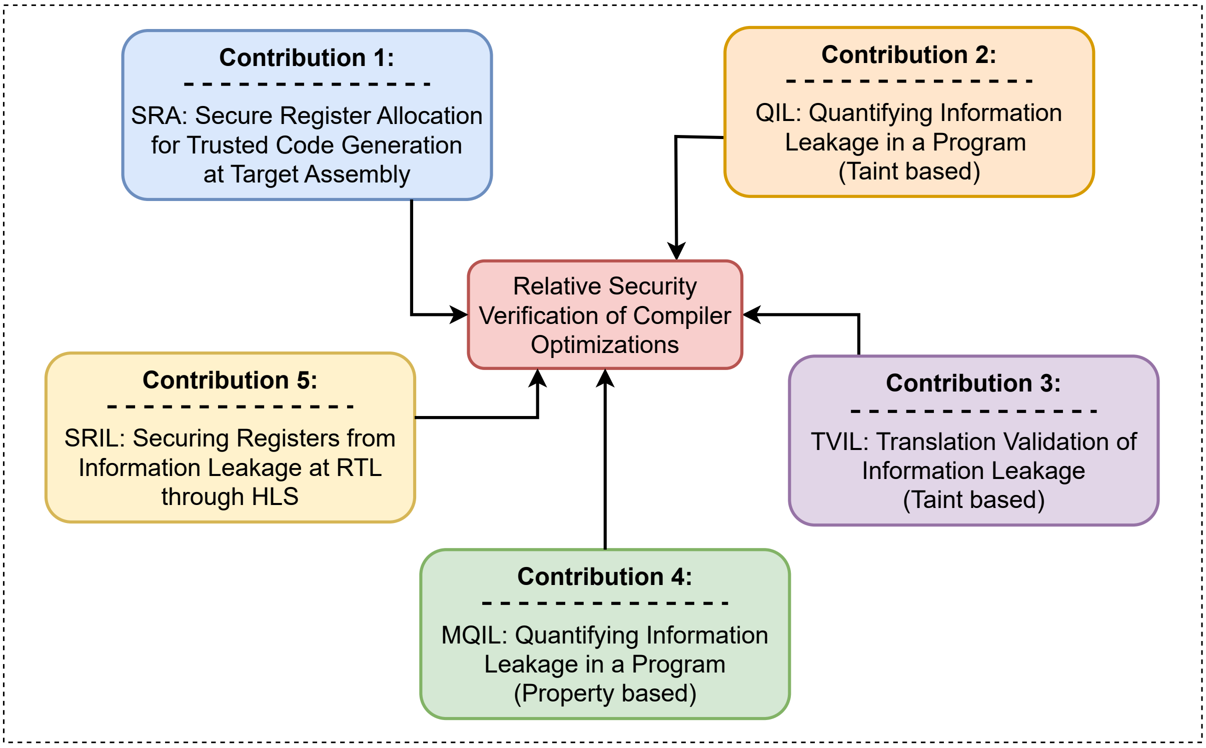

Contribution 1:
SRA: Secure Register Allocation for Trusted Code Generation at Target Assembly
Contribution 2:
QIL: Quantifying Information Leakage in a Program (Taint based)
Relative Security Verification of Compiler Optimizations
Contribution 5:
SRIL: Securing Registers from Information Leakage at RTL through HLS
Contribution 3:
TVIL: Translation Validation of Information Leakage (Taint based)
Contribution 4:
MQIL: Quantifying Information Leakage in a Program (Property based)

Fig. 1.6: *Contributions of the Thesis*

1.5.1 SRA: Secure Register Allocation for Trusted Code Generation

Register allocation (RA) is an essential optimization performed by a compiler. This thesis analyzes the security threat of RA concerning information flow. We show that RA is secure when there is no splitting and spilling into memory. We also show that register allocation with splitting is secure based on our attack model. Then, we show that RA can lead to information leaks during spilling as it introduces new leaks through memory. Further, our experimental results on various benchmarks show that RA in LLVM is leaky. To address this vulnerability, we propose a secure RA approach in LLVM that mitigates the risk of new leaks during spilling and generates a secure target assembly. Our experimental evaluation on 21 randomly chosen benchmarks shows the effectiveness of our proposed approach in terms of performance overhead. In the proposed secure register allocation approach, the average increase in total instructions, cycles, block RThroughput, and resource pressure are 2.09%, 1.44%, 1.41%, and 0.43%, respectively.

1.5.2 QIL: Quantifying Information Leakage for Security Verification of Compiler Optimizations

In this thesis, we attempt to quantify the information leakage in a program for the security verification of compiler optimizations. We demonstrate that static taint analysis is applicable for security verification of compile optimizations. We develop a completely automated approach for quantifying the information leak in a program in the context of compiler optimizations with respect to information flow. We proposed a leak propagation vector to capture the information leak at various program points. We introduced cutpoints in a program to efficiently analyze the implicit flow and propagate the leaks to subsequent paths to measure the overall information leak in a program. Our method avoids many false-positive scenarios due to implicit flow. It can handle leaks in a loop and finds a fixed point of leaks in loops. We propose three quantification parameters and verify the relative security of source and optimized programs, considering the optimization phases of a compiler as a black box. The soundness, termination, and complexity of the proposed approach are also presented. We use some Computation Tree Logic (CTL) properties to reduce the recursive calls and, thus, the overall complexity of the proposed quantification method. To check properties, we create a define-use version of the program on which properties are checked. This avoids state exploration problems during model checking. Our experimental evaluations in SPARK [68] show that the SPARK compiler is actually leaky, as it introduces new leaks during optimizations for 15 out of the total 20 cases. We present the results for the average execution time with respect to the lines of code and it shows that our implementation is linear for our taken benchmarks in the experiment.

1.5.3 TVIL: Translation Validation of Information Leakage of Compiler Optimizations

The third contribution of this thesis is to develop a translation validation of information leakage for the optimization phase of the compiler without considering any intermediate information from the compiler. The proposed method measures the information leakage in a path, uses a concept of leak propagation over paths and loops (proposed in our Contribution 2), and defines the relative security between the source and optimized programs. The proposed method bi-simulates both source and optimized programs at the path level and propagates the information leaks to the subsequent paths recursively for checking the relative security between the programs. Three different path-level relative security are formulated, namely secure paths, relatively secure paths, and conditionally secure paths. This thesis considers two attack models based on different observation points to access the local memory and proposes different translation validation approaches. We propose various look-ahead properties in CTL to reduce the overall complexity of our proposed method. The correctness, termination, and complexity of the translation validation method are also provided. The experimental results in the SPARK compiler on various benchmarks show that the optimized program is not relatively secure as it does not preserve source-level security in 13 out of the total 21 cases. The average speed-up of the translation validation method with property checking is 3.67X times compared to the same without property checking. We also show the linearity of our implementation concerning the code size and the execution time.

1.5.4 MQIL: Model Checking based Quantification of Information Leakage in a Program

The information leakage in a program due to compiler optimizations opens up side-channel attacks through which secret inputs like keys may leak via intermediate variables in cryptographic applications. In this thesis, we track the information flow in a program using a bounded model checker CBMC. To achieve this, we take our proposed information flow tracking concepts in Objective 2 and model the same as a set of assertions, which will be checked by CBMC. Conceptually, we create two copies of the original program and add assertions that check if a variable can have two different values in two copies of the program for different values of a sensitive input. The number of assertions failed by the CBMC tool is the overall information leakage of the program. We developed a practical tool for compilers like LLVM to quantify the information leakage that handles any C program. The proposed approach also verifies the relative security of LLVM IRs. To achieve this, we generate C code from the optimized IR using llvm2c [11] and then use the above method of quantifying information leakage in both source and optimized programs for checking relative security. In our experiment, we run the proposed approach for various cryptographic applications and show that it can successfully track information leakage. Our experiments reveal that LLVM introduces information leakage during optimizations as the optimized program is not relatively secure to the source program for three out of the four cryptographic benchmarks in our experiment. Our proposed method takes a negligible amount of time to quantify information leaks in cryptographic programs like Advanced Encryption Standard (AES). Thus, it is a practical and scalable method for the quantification of information leakage in a program.

1.5.5 SRIL: Securing Registers from Information Leakage at Register Transfer Level

The scan chain is used for the design for test (DFT) technique for testability improvement in sequential circuits [15,103]. Through the scan chain, the values of the internal registers of a design can be updated or retrieved at any point of execution. This scan chain of a design can be used as a side channel to retrieve the secret keys of a cryptographic implementation on hardware. This thesis proposes a method for protecting the registers from information leakage with the help of HLS and bubble pushing against such scan-based side-channel attacks. First, we perform a taint analysis on the behavioral specification to identify the variables that may potentially leak sensitive information explicitly or implicitly using our method proposed described above. We then mark the corresponding registers that store those leaky variables during HLS. In this thesis, we also introduce bubble-pushing across various RTL components like adder, multiplier, etc. Finally, we propose an RTL bubble-pushing algorithm starting from sensitive registers. This bubble pushing actually corrupts the register content and thus breaks the direct correlation between the cryptographic keys and the register content. Thus, even register access through a scan chain won't be useful to retrieve sensitive inputs. The average increase in area overhead is 3.86% and timing overhead is 4.58%. Thus, our proposed scheme has a negligible performance overhead. Moreover, our experiment on the AES based on the Test Vector Leakage Assessment (TVLA) shows that our proposed bubble-pushing approach protects against power side-channel attacks.

1.6 Organization of the Thesis

The thesis is organized into eight chapters. The contents of each of the eight chapters are summarized as follows:

- **Chapter 2:** This chapter provides the literature survey on the state-of-the-art related to this dissertation.

- **Chapter 3:** This chapter presents SRA, the method for securing register allocation with spilling and splitting.

- **Chapter 4:** This chapter presents QIL, the method to quantify the information leakage in a program for security verification of compiler optimizations.

- **Chapter 5:** This chapter presents TVIL, the translation validation method for information leakage of compiler optimizations.

- **Chapter 6:** This chapter presents MQIL, the approach to measure the information leakage and verify the relative security using the model checker CBMC.

- **Chapter 7:** This chapter presents SRIL, the protection of the registers from scan-based side-channel attacks.

- **Chapter 8:** Finally, this chapter concludes the thesis. The future perspectives in this domain are discussed in this chapter.

In each contribution chapter, the proposed algorithms are presented with adequate examples. The correctness and complexity of the algorithms are also presented wherever applicable. We have also presented detailed experimental results for each contribution.

2
Background and Literature Survey

In this chapter, we discuss the background on the security of compilers, followed by the literature related to the security of compiler optimizations. This survey aims to identify the gaps in the existing approaches for security verification in the context of compiler optimizations, which have been addressed in this thesis.

2.1 Target Level Attacks

In this Section, we discuss various target-level attacks apart from the information flow property discussed in Section 1.3.1 that are possible on an optimized or target code by a target-level attacker. To illustrate what attackers can do with compiled code, here we present examples of the most relevant threats a secure compiler needs to mitigate.

2.1.1 Confidentiality

To preserve the confidential property of the compiled code, it should be protected from unauthorized access to sensitive data. Consider the Java source code given in Fig. 2.1. The 'secret' variable is employed to store sensitive data and is isolated from other source-level code, ensuring its inaccessibility to potential attackers at the target level. In the target language, memory locations are typically identified by natural numbers, which means that an attacker could potentially ascertain the address where the ‘secret’ is stored. Through the process of dereferencing the numeric representation of the ‘secret’ location, attackers can compromise the code’s intended confidentiality property [12].

```
private int secret = 0;
public int setSecret() {
    secret = 1;
    return 0;
}
```

Fig. 2.1: *Confidentiality property violation*

2.1.2 Integrity

To preserve the integrity property of the compiled code, it should be protected from intentional unauthorized updates to sensitive data. The ‘proxy’ function in Fig. 2.2 assigns the value 1 to the ‘secret’ variable and subsequently invokes the ‘callback’ function. The ‘secret’ variable remains inaccessible to the code within the ‘callback’ function at the source level. However, if this code is compiled into a target language that permits manipulation of the call stack, it can potentially access and modify the ‘secret’ variable. Similarly, malicious target-level code can tamper with the return address stored on the stack, disrupting the expected flow of computation. This breach of security violates the integrity property [12] for the source code.

```
public int proxy( ) {
    int secret = 1;
    callback();
    return 0;
}
```

Fig. 2.2: *Integrity property violation*

2.1.3 Finite Memory Size

When dealing with memory management, the size of available memory can significantly impact the behavior of a component [73]. However, source languages often do not explicitly address memory size constraints. Consider a source language example shown in Fig. 2.3, which includes a dynamic memory allocation operation ‘new.’ The ‘kernel’ function allocates ’n’ new objects, invokes a ‘callback’ function, and executes security-critical code before returning 0. At the source level, the security-critical code will consistently execute. However, when this code is compiled into a language that confines memory to accommodate only ‘n’ objects, code execution can be disrupted during the ‘callback’ function. If the ‘callback’ function attempts to allocate an additional object, the execution of the security-critical code may be compromised and not executed as intended.

```
public int kernel(n) {
    for (i = 0 to n){
    new Object();
    }
    callback();
    // security-relevant code
    return 0;
}
```

Fig. 2.3: *Security violation by finite memory size*

2.1.4 Deterministic Memory Allocation

Dynamic memory allocation is a feature that introduces complexities [106], as demonstrated in the code depicted in Fig. 2.4. In this example, the code allocates two objects and subsequently returns the first one. While at the source level, ‘Object y’ remains inaccessible, this is not necessarily true in certain target languages. A target-level attacker with knowledge of the memory allocation order can predict the allocation location of an object and manipulate its memory contents. The attacker could tamper with ‘y’ by either making careful guesses or calculating its address.

2.1.5 Discussion

These source-level security properties discussed above are important to verify during compiler optimizations. However, this thesis targets the security of compiler optimizations in the information flow perspective (presented in Section 1.3.1) of a program. Specifically, we

```
public Object newObjects( ) {
  Object x = new Object();
  Object y = new Object();
  return x;
}
```

Fig. 2.4: *Security violation by deterministic memory allocation*

focus on checking the relative security of the optimized codes with respect to the source program. It may be noted that a target program may not be relatively secure to the source program due to the violation of other security properties.

2.2 Secure Compilation

Secure compilation refers to the process of transforming human-readable or high-level source code into machine-executable instructions while preserving the security properties of the software. These security properties primarily involve confidentiality, integrity, authenticity, and availability. The compilation process introduces potential vulnerabilities, including information leaks, control flow alteration, memory corruption, code injection points, and issues related to code optimization, etc. Secure compilation seeks to address these vulnerabilities and maintain the security goals by incorporating security-aware techniques and methodologies into the compilation process. This may involve type-based analyses, information flow tracking, proof-carrying code, language-based security, control flow integrity, formal verification methods, code transformation and obfuscation, and other security-enhancing measures.

The correctness-security gap of compiler optimizations has attracted some attention in recent times. The security impact of compiler optimizations identified in CWE-14 [3] and CWE-733 [4] discusses the dead store problem. Chen et al. [41] and Cuoq et al. [50] use compiler fuzzers, providing a practical approach to discovering compiler bugs. Test case reduction tools are applied to minimize the input size after discovering a bug [113]. Wang et al. [131] developed a STACK system that finds bugs arising from undefinedness optimizations. Their model detects the patterns of undefined behavior. Molnar et al. [97] developed a security model that detects and removes the control-flow side-channel attacks. Their security model relies on source-to-source transformations, usually exhibiting high performance and memory overhead.

Besson et al. [29] proposed a formal definition of information flow preserving (IFP) program transformations that consider the attacker's knowledge to determine the information leak of a program. They relate the knowledge of the attacker before and after the program transformation for the security validation of the transformation. The authors [30] then proposed a compositional proof principle for proving the information flow preservation of a transformation. The authors also show the automatic verification and mitigation of information-flow leaks that are introduced by dead-store elimination and register allocation optimizations using a translation validation technique. They model the information leak of a program in which attackers can observe the memory at some fixed observation points.

The authors in [100] developed a translation validation method for secure compilation. Their work expressed the security properties using automata, which operate over a bundle of program traces. They formulate the refinement relations in the context of secure compilation, which can be used to show that the security properties of a program are preserved after its transformation. However, there is no implementation proof of their methodology, which is also difficult for a real compiler.

A fully abstract compiler translates equivalent source-level components into equivalent target-level ones. Fully abstract compilation (FAC) preserves and reflects the equivalence of behaviors between the original and compiled programs in all untrusted execution contexts. The authors in [106] provided a FAC scheme that considers strongly typed object-oriented language as the source and untyped assembly language as its target.

In [109], authors explored a robustly safe compilation (RSC) criterion for proving the secure compilation. Their criterion ensures that the compiled code maintains the essential safety properties of the source program when exposed to any adversarial interactions with the compiled program. They claim that RSC has security guarantees and generates more efficient compiled code than FAC. The authors in [13] provided a comprehensive characterization of trace properties for correctly compiled programs, aligning with both the source and target properties. Their framework naturally accounts for various complexities, including undefined behavior, resource limitations, different value domains, side channels, and abstraction mismatches. Their approach extends to definitions of secure compilation, ensuring protection when linking compiled code with adversarial components. Recent approaches on secure compilation can be found in [55, 57, 58].

2.2.1 Discussion

Authors in [29, 30, 100] validated their secure compilation approach using the CompCert C compiler which is a compact formally developed compiler. The authors in [100] clearly mentioned that the mathematical proof for secure compilation is infeasible for compilers like GCC or LLVM with millions of lines of code. Moreover, the FAC and RSC proposed in [106] and [109] are targeting only assembly languages. Thus, secure compilation techniques are not feasible in practice for realistic compilers. In addition, the secure compilation schemes are computationally intensive and may not be scalable to large and complex software systems. The next section discusses the security analysis of various compiler optimization techniques in the literature.

2.3 Security Analysis of Compiler Optimizations

Researchers attempt to analyze the security of individual optimizations applied by a compiler and propose a secure version of the insecure optimizations found.

2.3.1 Dead Store Elimination

Deng and Namjoshi [52] proposed a leaky triple (discussed in Section 2.4.1) formulation of security and check the relative security of a program before and after transformation. The authors present a polynomial-time algorithm for secure dead code elimination and introduce a refinement relation for the security of other compiler transformations. Their approach allows the removal of a dead store to variable v under three scenarios as follows.

- *Scenario 1:* If v is post-dominated by other assignments to v, it is secure to remove it.
- *Scenario 2:* If v is untainted at the final location and v is untainted at the location immediately before a dead store, it is secure to remove it.
- *Scenario 3:* If v is untainted immediately before the dead store, no other assignment to v is reachable from the dead store, and the store post-dominates the entry or start node, then it is secure to remove.

The generated code does not ensure there is no dead store, as it may hamper the source-level security. Thus, it allows the elimination of only the dead stores, which does not affect the leakage of sensitive information.

2.3.2 Single Static Assignment

In [53], the authors propose a mechanism unSSA to restore the security of single static assignment (SSA) transformation. The unSSA ensures that the generated program after SSA is at least as secure as the source program. Their approach reverses the SSA transformation and assigns multiple versions of a variable with the same name to remove the exposure of new leaks through the intermediate values of the variable. They partition the variants of the variable into groups based on some properties and rewrite the name of every variable in the group with a new name, i.e., the name of the representative variable in the group.

2.3.3 Register Allocation

In [30], the authors investigate the information flow preservation of register allocation in the CompCert C compiler, which relies on register and variable mapping information. They perform a matching analysis at each program point during the execution of both the source program and the transformed program after register allocation. When there is a spilling to a memory location exists in the transformed program, their matching analysis returns there is no corresponding mapping with a source variable. To fix this information leak, they store zero in all those variables at the end of the program.

The authors in [59] protected the register spilling with respect to integrity and confidentiality. Their work focused on implementing a register spilling protector for the AArch64 backend of the LLVM compiler framework. Their implementation uses generalized ARM pointer authentication. Their register spill protector can identify the modifications done on the register spills and encrypt the memory content to ensure confidentiality, which renders the attackers without the knowledge of the original data stored in the memory.

2.3.4 Other Optimizations

D'Silva et al. [56] studied various compiler optimizations to identify the correctness-security gap. They highlight and formally analyze the role of compiler optimizations in introducing security vulnerabilities. The authors identified that the security gap could be analyzed in terms of observables, i.e., it requires more information about the internal state to be observable than that needed to analyze the correctness. The paper mentions several approaches to detect security violations. They conclude that the gap arises due to the techniques that do not model the state of the underlying machine. The difficulty of checking security has a further impact on translation validation.

The SWIPE algorithm [64] performs a source-to-source transformation that adds additional instructions to erase potentially sensitive data immediately after its use, thereby improving security at the source level. Their approach aims to reduce the lifetime of sensitive variables by performing static analysis to identify the locations for adding the erase instructions in the original program. However, the effectiveness of SWIPE may be limited after compiler optimizations have been applied.

2.3.5 Discussion

Register allocation is a mandatory transformation for any source program to generate the machine code. Thus, it should be properly investigated from the security point of view. Authors in [30] consider the register variable mapping information for security analysis of the register allocation. However, in modern compilers like LLVM, obtaining such mapping information is challenging due to variable renaming and temporary variable introduction in the IR. Therefore, our analysis of the security of register allocation does not rely on any mapping information from the compiler in Chapter 3. Although register allocation is shown as not an IFP in [30], the security of register allocation in all aspects, considering spilling and splitting, has not been investigated yet. Moreover, none of the existing works provide a secure register allocation algorithm. The authors in [59] targeted the AArch64 backend to secure the register spilling. In Chapter 3, we have analyzed the security issues of register allocation with spilling and splitting. We have also provided a secure register allocation scheme for a widely used compiler targeting the x86_64 CPU architecture.

Security analysis of individual compiler optimization techniques is not feasible in practice since a compiler applies hundreds of optimizations to improve code performance. Moreover, the exact optimizations required for a program to enhance the target code are unknown. Thus, the user cannot apply the required optimizations manually to preserve the source-level security. It is also possible that some of the secure optimizations may or may not be applied by a compiler. Therefore, we need a precise analysis for relative security verification in a program considering the applied optimizations as a black box. This thesis aims to verify the relative security between the source and optimized programs using a translation validation approach which has been addressed in Chapters 4 to 6.

2.4 Security Measurement Approaches

The existing works on securing individual compiler optimizations use primarily two security measurement approaches: leaky triple notion (non-interference) and taint analysis to verify the security with respect to information flow in a program. We now discuss these approaches in detail with their pros and cons in the following.

2.4.1 Leaky Triple Notion (Non-interference)

In a program, variables can be partitioned into high-security (sensitive) or low-security (non-sensitive) types. Program variables or state variables are always of low-security type. Input variables can be of high or low-security type. The inputs and outputs are the same for both the source and transformed programs. Therefore, the security types of the inputs are the same for both source and transformed programs. The information leakage of a program using *leaky triple notion* in [52] is defined as follows:

Definition 2.4.1 (Information Leakage)**.** *Let us have three values, namely a, b, and c, where a and b are the values of a high variable such that $a \neq b$ and c is a low variable value. There are two input pairs (a, c) and (b, c). If, for a program S, the computation of S on both the input pairs either differs in the sequence of output values or the value of one of the low variables differs at their final states when both terminate, then program S is said to leak information and (a, b, c) is called as a leaky triple for program S. We have taken this definition from [25].*

The *correct transformation* is defined as follows: Program transformations do not alter the set of input variables. A transformation from program S to program T may alter the operations and control structure of S or the set of state variables. The transformation is correct if, for every input value a, the sequence of output values for executions of S and T from the input value a is identical.

Secure Transformation with respect to Leaky Triple: An optimized program T is said to be secure if all the leaky triples that belong to T also belong to the source program

S. It ensures relative security, i.e., T is as secure as S. It essentially ensures that compiler transformation has not introduced any new leak in the optimized program, i.e., the optimized program is not more leaky than the source program. Note that it does not guarantee the security of the source program. The source program may be more leaky than the optimized program. Secure information flow has been discussed in detail in [27, 54].

2.4.1.1 Problem with Leaky Triple

Suppose that the transformation from S to T is correct. Consider a leaky triple (a, b, c) for T. If the computations of T from inputs $(H = a, L = c)$ and $(H = b, L = c)$ differ in their output from correctness, this difference must also appear in the corresponding computations in S. Hence, the only way in which T can be less secure than S is if both computations terminate in T with different values for low variables while the corresponding computations in S terminate with identical values for low variables.

```
void foo()
{
      x = password();
      ... //use x;
      y = x;
      ... //use y;
      x = 0; //dead store
}
```

(a)

```
void foo'()
{
      x = password();
      ... //use x;
      y = x;
      ... //use y;

}
```

(b)

Fig. 2.5: *An Example of Dead Store Elimination (DSE): a) Source code, b) After DSE*

The function *foo()*, in Fig. 2.5 reads a password, and later it is cleared from memory after its use. After DSE, the statement ‘x = 0’ is removed as it is a dead store. The leaky triple notion considers both the programs in Fig. 2.5 as the same secure because both leak the same password. But in reality, this is not true because before DSE, the password leaks through only low variable y, whereas after DSE, it is leaked through both x and y. Thus, the leaky triple notion cannot always quantify the leakage. It considers a more leaky program as same secure as the source.

The relative security of two programs, S and T, cannot be guaranteed by assuring that if a leaky triple belongs to T, it also belongs to S. This issue can be resolved by the following approach to ensure the relative security of T and S. If a high input gets leaked in T, it must leak in S irrespective of the low variables. Moreover, generating a leaky triple through the leaked high input in T is always possible. Nevertheless, we cannot guarantee the same leaky triple would also be there in S due to the fact that both programs may not have common low variables as new temporary variables are introduced during the optimizations. In contrast, it is not required to ensure the same leaky triples belong to both T and S because if an attacker can generate a leaky triple for T, there would always be a possible leaky triple for S as well through the leaky high input irrespective of the low variable. Thus, leaky triple is not a feasible approach to verify the relative security between the source and optimized programs.

2.4.2 Taint Analysis

In a program, taint analysis [37] is the process of finding the taintness of each and every variable at any program location. Taint is a boolean set: $\{T$: tainted, U: untainted$\}$. Using taint analysis, we can track the information flow attacks as discussed in Section 1.3.1 and Section 1.3.2. A taint environment is a function t_e: $Variable \rightarrow Taint$. It assigns a taint value to each program variable name. The taint environment t_e can be extended in the following way:

1. Constant: Each constant is considered as untainted.
   $$t_e(c) = U, \text{ if } c \text{ is a constant.}$$

2. Variable: The taint value of variables is obtained from its current taint value.
   $$t_e(v) = t_e(v), \text{ if } v \text{ is a variable.}$$

3. *input*(): The return value of the *input*() is input sensitive. So, it is tainted.
   $$t_e(input()) = T$$

4. Operator with T: If one operand x is tainted for an assignment operation of the form $v' = x < op > v$, the taint of left hand side variable v' is tainted,
   $$t_e(T \ < op > \ v) = T$$
   if v is a variable in an expression and op is a binary arithmetic operator.

5. Operator with U: If one operand x is untainted for an assignment operation of the form $v' = x < op > v$, the taint of left hand side variable v' depends on the taintness

of the other operand v,

$$t_e(U \ < op > \ v) = t_e(v)$$

if v is a variable in an expression and *op* is a binary arithmetic operator.

6. Expression: The taint type of an expression can be obtained from the taint types of each subexpression,

$$t_e(v_1 \ < op > \ v_2 \ldots < op > \ v_n) = op_{i=1}^{n} t_e(v_i)$$

if $v_1, \ v_2, \ \ldots \ v_n$ are variables in an expression and *op*s are binary arithmetic operators.

7. Assignment: The taintness of the assigned variable changes according to the taintness of the right-hand side expression.

8. Conditional: If the condition is tainted, all the assignments made in both branches will also be tainted because of control dependency; otherwise, the taintness is obtained from data dependency only. This is basically over-approximating the tainting of variables.

9. While loop: If the condition is tainted, all the assignments in the branch will also be tainted; otherwise, the assignments may get tainted in further iterations.

Here, we discussed the taint analysis in brief. The detailed analysis is given in [37]. Now, we discuss the dynamic and static taint analysis in detail below.

2.4.2.1 Dynamic Taint Analysis

Dynamic taint analysis [24, 47, 51, 75, 120, 130, 134] computes the taintness of variables at run time. Since it considers a single execution trace, it has all available information during the computation of variable taintness. However, the coverage analysis is very weak in dynamic taint analysis, and some information leaks may be completely ignored. Moreover, generating test cases with high coverage is a non-trivial task. Also, it fails to provide any leak information or taint flows for the code which is not executed. Thus, researchers supplement the static analysis for dynamic taint analysis approaches as well for various applications [20, 135]. Since our objective in this thesis is to identify the leakage at compile time, dynamic taint analysis is not applicable to our purpose.

2.4.2.2 Static Taint Analysis

Static taint analysis computes the taintness of variables at compile time. Thus, it takes all the execution traces of a program. Many static taint analysis approaches in the literature consider only explicit flows in a program, i.e., it under-approximates the leak [19, 66, 88]. This leads to false negatives by ignoring the implicit flows in the program.

Authors in [37] detected a taint dependency sequence for security vulnerabilities using static taint analysis. However, their approach over-approximates the leak due to implicit flows. They consider a single path inside a loop while finding the fixpoint. A static information flow analysis is performed in [91] to handle implicit flows. The authors studied the effects of implicit flows over explicit flows and introduced the implicit edges in the flow graph. They have illustrated the usage of their approach on three different applications: security violation detection, type inference, and the effects of shared-thread variables on thread-local variables. They conclude that implicit flow detects additional violations in these applications. However, both approaches in [37, 91] over-approximating the implicit leak. Also, they have ignored the security violations due to loops in the program.

Authors in [34] studied various static analysis tools [19, 66, 88] that detect information flow for Android applications and concluded that these tools are suffering from under-tainting by ignoring the implicit flows. A static taint and initialization analysis approach called STILL developed in [129] which detects exploit code such as self-modifying code and indirect jump in web services. Another static analysis approach called EdgeMiner [36] is proposed for the Android framework, which automatically detects the implicit control flow transitions by generating API summaries. Their approach performs an inter-procedural backward data flow analysis to detect the implicit flows. Authors in [119] proposed a generic framework to track explicit flows by taint checking. They studied a few existing tools to show the soundness by their approach of explicit secrecy.

The state-of-the-art works proposed different taint analysis techniques for different applications, either statically or dynamically. Recent works also propose hybrid techniques considering both static and dynamic approaches to detect taint flows. Hybrid techniques use a dynamic supplement to static analysis [136] for their approach or a static supplement to dynamic analysis [20] to avail the combined benefit of both approaches. Since our objective is to identify the leakage at compile time, dynamic and hybrid taint analysis techniques are not applicable to our purpose.

2.4.3 Discussion

Although the static taint analysis has wide applications, none of the existing work has targeted compiler security verification. In some cases [52,53], existing static taint approaches [27, 54] are used to analyze a specific compiler optimization. However, such static taint analysis solutions have over-tainting problems, as discussed above. Also, no existing works give a detailed method to detect information leaks due to both explicit and implicit flows considering multiple paths inside a loop. Therefore, our second objective in the thesis is to develop a static taint analysis to quantify the leak in a program. The method should consider the possible impact of code optimizations.

There are some efforts in making individual transformation secure in the literature. However, there is limited study on identifying the relative security of the overall compiler optimizations considering the applied optimizations as a black box. Moreover, none of the literature considers the combination of explicit flows, implicit flows, and the fixpoint of a loop in an individual program for translation validation. In Chapter 4 and Chapter 5, we attempt to validate the information leakage after compiler optimizations, considering all possible leaks in a program. Finally, the approach is implemented on a modern-day compiler to validate the relative security between the source and optimized programs.

None of the existing works in the literature attempt to quantify and verify the information leak using property-based testing on a model checker. Thus, in Chapter 6 we aim to achieve the same by modeling the security properties as assertions in the input program. Finally, execute the program on a model checker to quantify and verify the relative security of compile optimizations.

2.5 Side-channel Attacks through Scan Access

The side channel attacks include timing analysis [32], differential power analysis [80], and scan-based attacks [49, 89, 118]. Here, we first discuss the purpose of a scan chain and then the related approaches to scan-based attacks and their countermeasures.

A scan chain is inserted into the design after the generation of the RTL for testing purposes. An example of a scan chain is shown in Fig. 2.6. Three registers, $R1$, $R2$, and $R3$, are connected in the scan chain as flip-flops FF. When the scan enable signal SE is 1, it takes input from the scan in port SI and generates output through the scan out port SO. Otherwise, the circuit takes input from the primary input PI and generates the output

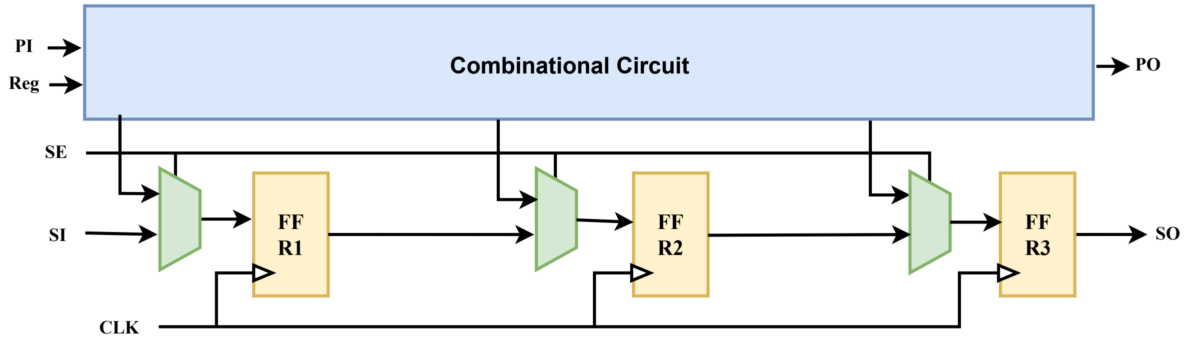


Fig. 2.6: *An Example of a Scan Chain*

through the primary output port PO. Using SI, the test engineer can set a fixed value to the registers in n-cycles (where n is the scan chain length). Next, he runs the design with SE=1 for one cycle. The registers will be updated with the values from the design. He then takes out those values through SO in n-cycles (by keeping SE=1). He then compares the design outputs with the expected outputs to identify any bug in the internal circuit due to the manufacturing process.

The attacker can utilize this scan chain of IC to get sensitive internal information of the design. To protect the ICs from scan-based side-channel attacks, researchers primarily proposed countermeasures to protect the scan chain with additional keys. Authors in [82] propose a Lock-and-Key technique to secure the design from scan-based side-channel attacks. Their lock and key technique divides the scan chain into subchains of equal length and randomizes the access to these subchains for the untrusted user. Authors in [49] show that obfuscating the scan chain order does not preserve security. Attackers can break this protection if they know the total states in the scan chain. Authors in [78] propose a dynamic scan-locking technique called Encrypt Flip-Flop. They encrypt the output of some flip-flops by adding MUXes at their outputs. The added Mux acts as a key gate, connecting the key to the select line of the Mux. Their technique prevents unauthorized access to scan data. Recently, authors in [126] have proposed a method to detect and prevent scan attacks to secure cryptographic chips with a test key. It can prevent both differential and signature scan-based attacks. In [89], the authors propose an attack to break the scan data protection in [78]. The authors in [89] analyze the pseudo-random number generator (PRNG) circuit to get the original seed used by [78] to generate obfuscated scan data. Recently, authors have proposed an SAT attack called ScanSAT in [18] to extract the key by extracting the logic-locked version of the scan obfuscated circuit.

2.5.1 Discussion

All the existing works protect the scan chain by obfuscating the scan access with additional key bits. Moreover, those techniques are vulnerable to various attacks as shown above. The real issue is the leakage of sensitive information through the registers, which enables side-channel attacks. However, none of the existing work targets obfuscating or corrupting the secret data passes from the circuit to the scan chain. In Chapter 7, we propose a new way to protect against information leakage through registers by making the data statistically independent of the secret inputs without obfuscating the design with additional keys.

2.6 Conclusion

In this chapter, we discussed several state-of-art literature on securing individual compiler optimization techniques. We presented the existing security measurement methods in the literature. We identified the limitations with the existing security measurement methods. We also discussed various scan-based attacks and their defenses at the register transfer level during HLS. In the subsequent chapters, we present the security analysis of register allocation, which is a mandatory step in a compiler, quantifying the information leak, verification of relative security by translation validation approach, a practical tool for quantification of information leak, and bubble-pushing approach for securing the registers at RTL.

3

SRA: Secure Register Allocation for Trusted Code Generation

3.1 Introduction

Register allocation (RA) is the mapping of program variables to a fewer number of machine registers [38,39]. Its objective is to minimize register usage by mapping one or more variables to a single register if their lifetimes are non-conflicting. In addition, RA also applies splitting and spilling. A variable is chosen to be split such that the non-conflicting live ranges of the variable can mapped to different registers for better register usage. However, due to a limited number of registers, it is not possible to map all the variables to registers. Thus, few variables are mapped into memory, which is called spilling. In this thesis, we investigate the security implications of RA with and without splitting and spilling. We explore the potential vulnerabilities to establish a secure compilation process that does not introduce any security weaknesses through RA. Specifically, this thesis attempts to answer - Does RA retain the security of the source program with respect to information flow in our following attack model?

Threat Model: In our attack model, we assume the attacker has no access to the secret data but has access to the executable binary of the source code and, thus, cannot modify but execute the source code. The ultimate objective of an attacker is to retrieve secret data. The attacker is aware of the implementation. Thus, he can control the other non-sensitive inputs to obtain the sensitive inputs from the intermediate values. The attacker has no access to registers directly but gains access to the memory at the end of the execution to achieve their goal. It is quite impossible to access the memory contents during the execution of a program. Similar attack models have also been considered in [29]. The detailed discussion of the threat model is presented in Section 3.7.

This thesis analyzes the security vulnerability of RA in the presence of splitting and spilling. Specifically, the contributions of this chapter are as follows:

- We show that RA is secure without spilling and splitting.
- We show that RA with splitting is secure in our attack model.
- We also show that RA with spilling introduces new information leaks through memory.
- Our experimental results using various benchmarks found that RA in LLVM is leaky.
- A secure RA in LLVM is proposed to avoid introducing new information leakage due to spilling.
- A detailed evaluation of the performance overhead of the proposed secure RA is presented.

The rest of the chapter is organized as follows. Section 3.2 provides the background of RA in detail. Section 3.3 discusses the concept of relative security between two programs. Section 3.4 describes our security analysis of RA. Section 3.5 proposes our secure RA scheme. Section 3.6 presents experimental results. Section 3.7 discusses the relevance of the attack model. Finally, Section 3.8 concludes the chapter.

3.2 Register Allocation

Let us assume that V_i is the set of variables with i number of variables in a program and R_j is the set of registers with j number of registers in the target architecture. Register allocation is a function that maps the set of variables to the set of registers and/or memory locations, $f : V_i \rightarrow R_j \cup M_k$ where usually $j \leq i$ and M_k is the reserved memory locations for spilled variables V_k where $0 \leq k \leq i$. The following scenarios may arise due to register allocation.

- A variable v_a is mapped to a register r_x, i.e. $f(v_a) = r_x$.
- More than one variable is mapped to a register. For example, $f(v_a) = f(v_b) = f(v_c) = r_x$ means the variables v_a, v_b and v_c are mapped to the register r_x.
- A variable is mapped to more than one register, which is called the live range splitting. In this case, some re-definitions of v_a are renamed first based on the non-conflicting live ranges, and then different instances of v_a are mapped to different registers, i.e., v_a is renamed to v_{a1}, v_{a2}, v_{a3} and mapped to registers r_x, r_y, and r_z, such that $f(v_{a1}) = r_x$, $f(v_{a2}) = r_y$, and $f(v_{a3}) = r_z$.
- If there are not enough registers, one or more non-overlapping variables needs to be spilled into memory, i.e., $f(v_a) = f(v_b) = f(v_c) = m_a$ where m_a is the reserved memory location for the non-overlapping variables v_a, v_b, and v_c.

We now discuss the usage of live range splitting and spilling below.

3.2.1 Live Range Splitting

A variable maps to more than one register for better register usage. The live range of the variable has been split into multiple ranges. The RA process considers each live range as a different variable and assigns registers in the best possible way. Consider an example of RA in Fig. 3.1. The original source snippet S with three variables a, b, and c are presented in Fig. 3.1(a). As per the live variable analysis on S, the three variables are live at the same time. The conflict graph for the same is represented in Fig. 3.2(a). Its clear that at least three registers are required to map the three variables. Let's assume there are three free registers, and the variables a, b, and c are mapped to registers $r1$, $r2$, and $r3$, respectively. The S after RA is presented in Fig. 3.1(b). Now, let us assume we have only two free registers available. Then, the question is, can we map these three live variables to two registers with splitting? Let us now split the line range of variable a to $a1$ and $a2$. After splitting a, the source code has been presented in Fig. 3.1(c). Only two variables are live at the same time if we perform a live range analysis on S after splitting. The conflict graph for the same is represented in Fig. 3.2(b), which shows that two registers are enough for mapping the variables. Let $a1$ and c are mapped to register $r1$, and $a2$ and b are mapped to register $r2$. The RA after splitting on s has been presented in Fig. 3.1(d). This example of RA shows how splitting helps to improve the register usage, i.e., the same source snippet S can be mapped to two registers with splitting instead of three registers without splitting. Now, another question arises: does live range splitting always improve register usage?

```
secure(b,c)
{
        a = read_pw();
        b = a + b;
        a = c + b;
        a = a + c;
        return a;

}
```

(a) *Source snippet S*

```
secure(b,c)
{
        r1 = read_pw();
        r2 = b, r3 = c;
        r2 = r1 + r2;
        r1 = r3 + r2;
        r1 = r1 + r3;
        return r1;
}
```

(b) *S after RA*

```
secure(b,c)
{
        a1 = read_pw();
        b = a1 + b;
        a2 = c + b;
        a2 = a2 + c;
        return a2;
}
```

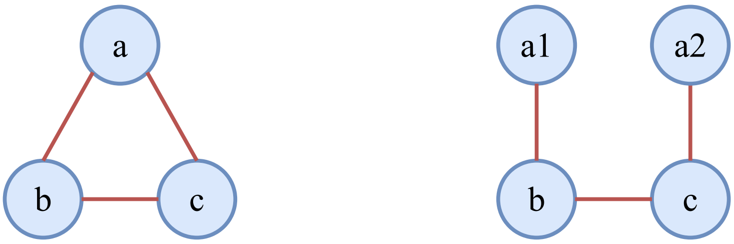
(c) *S after splitting*

```
secure(b,c)
{       r1 = read_pw();
        r2 = b;
        r2 = r1 + r2;
        r1 = c;
        r2 = r1 + r2;
        r2 = r2 + r1;
        return r1; }
```

(d) *S after RA with splitting*

Fig. 3.1: *An example of register allocation with and without splitting*

The live variable analysis on another source snippet S presented in Fig. 3.3(a) says that the three variables are live at the same time. For this source snippet S, the conflict graph is the same as represented in Fig. 3.4(a). Thus, the live range of variable a has been split into $a1$ and $a2$ to improve the register usage, and the S after splitting has been presented in Fig. 3.3(b). The conflict graph for the same is represented in Fig. 3.4(b). Its clear that at least three registers are required to map the four variables $a1$, $a2$, b, and c, as three variables are live at the same time. Let's assume there are three free registers, and the variables $a1$, b, and c are mapped to registers $r1$, $r2$, and $r3$, respectively, and $a2$ can be mapped to any



(a) *for S in Fig. 3.1(a)* **(b)** *for S after splitting in Fig. 3.1(c)*

Fig. 3.2: *Conflict graph showing improvement on register usage with splitting*

of the three registers. Here, $a2$ is mapped into register $r3$, and S after RA with splitting is presented in Fig. 3.3(c). From this motivational example its clear that splitting is always not beneficial.

3.2.2 Spilling

When splitting is not beneficial or the number of registers available is not sufficient to store all the variables, spilling is applicable. Spilling adds store instruction after the definition of the spilled variable to store the content into memory and load instruction before the use of the spilled variable to bring it back from memory for further use in the program. In Fig. 3.3(c), with two available registers, we need to spill one or more variables into the memory. Let's assume a is mapped into memory location m. The S after RA with spilling is presented in Fig. 3.3(d). The variables b and c are mapped into registers $r1$ and $r2$, respectively. The store instruction is added after the spilled variable a, and the load instruction is added before the use of a in Lines 3 and 6, respectively. This shows that with spilling, the RA is successfully performed with two registers. The security analysis of register allocation with splitting and spilling is presented in Section 3.4.

3.2.3 Impact of Register Allocation in Control and Data Flow

In the process of register allocation (RA), a source program is transformed into a program that utilizes registers and memory to improve performance. For the source code S in Fig. 3.3(a) the RA with spilling is presented in Fig. 3.3(d). The transformed program, which is produced by RA, is functionally equivalent to the source program. The control and data dependencies of the source program are not altered by the RA step. The RA step only

```
secure(b,c)
{  a=read_pw();
   b=c+b;
   a=a+b;
   return a;
}
```

(a) *Source snippet S*

```
secure(b,c)
{  a1=read_pw();
   b=c+b;
   a2=a1+b;
   return a2;
}
```

(b) *S after splitting*

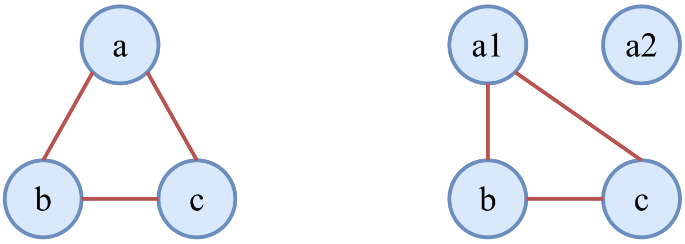
```
secure(b,c)
{
   r1=read_pw();
   r2=b,r3=c;
   r2=r3+r2;
   r3=r1+r2;
   return r1;
}
```

(c) *S after RA with splitting*

```
secure(b,c)
{  r1=read_pw();
   store r1,m;
   r1=b,r2=c;
   r1=r2+r1;
   load m,r2;
   r2=r2+r1;
   return r2; }
```

(d) *S after RA with spilling*

Fig. 3.3: *An example of register allocation with spilling and splitting*



(a) *for S in Fig. 3.3(a)* **(b)** *for S after splitting in Fig. 3.3(b)*

Fig. 3.4: *Conflict graph showing no improvement on register usage with splitting*

renames the variables with corresponding registers. The inputs, outputs, and their security types remain unchanged in both source and transformed programs.

Consider the example in Fig. 3.5, which represents the control/data flow of a source program S and its transformed program T after register allocation. The variable v at node n_k in Fig. 3.5(a) is mapped to register r at the corresponding node n_k in Fig. 3.5(b) where $1 \leq k \leq 9$. Let one of the inputs to variable $v3$ and register $r3$ is high input h (sensitive). It is possible to identify the propagation of the taint value of input h to r in T by applying taint analysis, which implies the existence of a control and/or data flow path from h to r

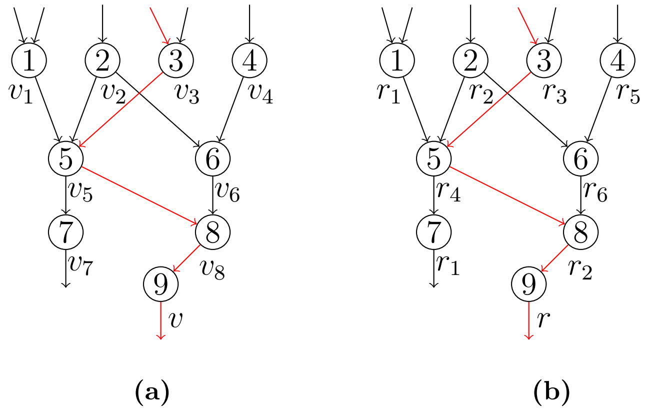


Fig. 3.5: *Control and data flow of (a) source program S, (b) after register allocation T*

that facilitates the propagation of the taint value. An illustration of the scenario is given in Fig. 3.5(a) and 3.5(b), where the edges in red represent the data flow of a path in the source program and its corresponding path after register allocation, respectively. Consider variable v in S is mapped to register r in the final states, and as S and T are functionally equivalent, there must exist a control or data flow path from input h to v in S. Consequently, the taint value of h must be propagated to v in S via this path. It illustrates the importance of taint analysis in detecting tainted data flow without explicit mapping information. The control/data flow of the original program remains unchanged in the transformed program after register allocation.

3.3 Relative Security

In a program, inputs can be partitioned into high-security (sensitive) or low-security (non-sensitive) types. Program variables are always of low-security type. The inputs and the outputs are the same for both source and transformed programs. Therefore, the security types of the inputs are the same for both source and transformed programs.

To define the relative security between two programs, we need to identify the information leakages in a program. The *taint analysis* is the primary method to identify the information leakages in a program using information flows. Taint analysis checks the explicit and implicit influence of sensitive inputs on program variables, violating the security properties of the program based on information flow.

Definition 3.3.1 (Relative Security)**.** *A transformed program T is said to be relatively secure to its source program S at the end of execution if it satisfies the following necessary condition followed by the sufficient condition. The necessary condition is if a high input h is leaking in T, it must be leaking in S. The sufficient condition would be for each of the leaky high inputs in T, the number of variables leaking a high input h in T must be not more than that of in S.*

```
void foo()
{ x=password();
   ... //use x;
  y=x;
   ... // use y;
  x=0;//dead store
}
```

(a)

```
void foo()
{ x=password();
   ... //use x;
  y=x;
   ... // use y;

}
```

(b)

Fig. 3.6: *An Example of Dead Store Elimination*

Here, we defined the relative security at the abstract level and it is discussed in Chapters 4 and 5 in more detail. Let's analyze the relative security between the programs before and after DSE in Fig. 3.6. The necessary condition holds as the password is leaking in both programs. However, it violates the sufficient condition as the password leaks through two variables (x and y) in the transformed program but through one variable (y) in the source program. Thus, according to our definition of relative security, DSE is not relatively secure.

To verify the relative security of register allocation, we use the taint analysis (presented in Chapter 4) to identify the leaky variables in the source program S. Similarly, we perform a taint analysis on the transformed program T to identify the leaky registers in T. However, few tainted variables are mapped to memories when there is no free register available. Thus, these memory locations are also marked as leaky or tainted in T. We then compare the leaks between these two programs to decide the relative security.

3.4 Security Analysis of Register Allocation

We assume that the register allocation has already proved functionally correct [102,115,116], and we analyze the security of the transformation after register allocation with respect to information flow. We analyze the security issues in RA with and without splitting and spilling. In the following, we consider three scenarios of RA to analyze their security separately;

1. *Scenario 1 (RA without splitting and spilling):* Each variable is mapped to a register, and no memory location is used.

2. *Scenario 2 (RA with splitting and without spilling):* Some variables mapped to more than one register.

3. *Scenario 3 (RA with spilling and without splitting):* Some variables mapped to memories instead of register.

Let the source program before register allocation be denoted as S, and let the transformed programs in three different scenarios be denoted as T_1, T_2, and T_3. Our primary goal is to determine the relative security of each transformed program with respect to S. Specifically, we seek to answer the following questions: (i) Is T_1 relatively secure to S? (ii) Is T_2 relatively secure to S? (iii) Is T_3 relatively secure to S?

Scenario 1: $T1$ is leaky if i) a high input h leaking in $T1$ but not leaking in S or ii) the number of registers leaking a high input h in $T1$ is more than that of in S. We will verify the necessary conditions followed by the sufficient condition of relative security. Each variable of S is mapped to a register in $T1$. Thus, if a high input h is leaking through r in $T1$, it must be leaking through its corresponding variable v in S. So, it holds the necessary condition of relative security. Now, we need to verify the sufficient condition. It is possible that more than one variable of S is mapped to a single register r in $T1$. Thus, if h is leaking through x number of registers in $T1$, then the corresponding variables of those x registers in S should not be less than x. Let variables $v1, v2, v3$ mapped to register $r1$ and variables $v4, v5, v6$ mapped to register $r2$. This means that at the end of the program, $r1$ and $r2$ have the content of $v3$ and $v6$ (last assigned variables), respectively. Assume high input h leaks through both registers $r1$ and $r2$. Then h must leak through at least two variables $(v3, v6)$, i.e., the corresponding variables mapped last to registers $r1$ and $r2$. Thus, the sufficient condition of relative security always holds as two registers leaking h in $T1$ and two variables leaking h in S. *Thus, T_1 is relatively secure to S.*

Scenario 2: A variable may be mapped to more than one register in $T2$. Thus, when there is a leak of high input h through a register, there may not be a corresponding variable

v leaking h in S since v has been split. The source S in Fig. 3.3(a) after splitting variable a is represented in Fig. 3.3(b). The password (h) in Fig. 3.3(c) is leaking through the register $r1$ after splitting, but the corresponding variable a is not leaking the password as it has been overwritten in Fig. 3.3(a). Thus, in scenario 2, h is leaking in $T2$ but not leaking in S. This violates the necessary condition of relative security. So, $T2$ is not relatively secure to S.

The generic security analysis of splitting is as follows. The two definitions of variable a are renamed to $a1$ and $a2$. Here, the focus is to check the leak through variable $a1$ for the relative security of splitting. It may be noted that the leak through $a2$ does not impact the relative security due to splitting. The variables $a1$ and $a2$ can be either tainted or untainted. Thus, there are four possibilities for the two definitions of a based on the taintness. Here there are three cases arise for relative security. i) If $a1$ is untainted, the leak of $a1$ does not impact the relative security irrespective of the taintness of $a2$. ii) If $a1$ is tainted, and $a1$ and $a2$ are mapped to the same register $r1$, then the original definition of $a1$ gets overwritten by $a2$ and $r1$ in both the source program and after register allocation, respectively. In this case, splitting does not introduce new security vulnerabilities. iii) If $a1$ is tainted, and $a1$ and $a2$ are mapped to different registers $r1$ and $r2$, respectively, then there is a leak through $r1$ if $r1$ is not overwritten by any other mapped variable till the end of the execution of the program. Thus, splitting is leaky if the attacker has access to registers. However, in our attack model, we assume the attacker has no access to registers. So, h cannot leak in $T2$. Thus, the condition of relative security holds as there is no leak in $T2$. *Thus, T_2 is relatively secure to S based on our attack model.* However, splitting is leaky if we assume that register content is also accessible in a different attack model. In such a case, a secure splitting algorithm needs to be developed.

Scenario 3: A variable v in S may be mapped to a memory m in $T3$ due to spilling. When sensitive data is spilled to memory location m in $T3$, there would be no corresponding variable in S for m. The password is leaking through memory location m in Fig. 3.3(d), but there is no leak of the password in source S in Fig. 3.3(a). So, when there is a leak through a spilled memory location in $T3$, there is no leak in S. This violates the necessary condition for relative security in Definition 3.3.1. *Thus, T_3 is not relatively secure to S.*

We, therefore, consider only scenario S3, i.e., register allocation with spilling is not secure based on our attack model. In this thesis, we focus on developing a secure spilling algorithm. Specifically, we consider the spilling in the LLVM compiler and enhance it to a

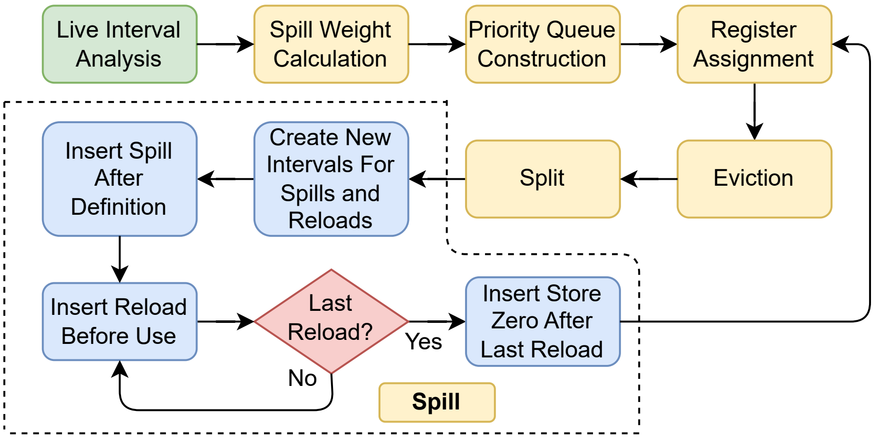

Live Interval Analysis
Spill Weight Calculation
Priority Queue Construction
Register Assignment
Insert Spill After Definition
Create New Intervals For Spills and Reloads
Split
Eviction
Insert Reload Before Use
Last Reload?
Yes
No
Insert Store Zero After Last Reload
Spill

Fig. 3.7: *Secure Greedy Register Allocation in LLVM*

secure version. The intuition is to erase the sensitive content of the spilled memory location after its last use. The secure register allocation is discussed in the next Section.

3.5 Securing Register Allocation in LLVM

This Section discusses our proposed secure register allocation approach in LLVM. There are four different register allocators (RA) in LLVM, namely, *basic*, *fast*, *greedy*, and *PBQP*. We found all the register allocations are leaky. This thesis proposes a secure greedy register allocation method in LLVM. We chose the greedy method since it is the default one in LLVM. However, the same approach can be applied to other methods as well. In this thesis, we show how to secure the Greedy Register Allocation (GRA) of LLVM. The steps for the secure greedy register allocation in LLVM are presented in Algorithm 1.

The compiler performs live interval analysis of all the program variables to allocate them to registers. The GRA finds the spill weights of all live intervals based on heuristics such as the number of uses, conflicts, etc. A priority queue Q is constructed based on *allocation priorities* of live intervals of the program variables. The queue allocation priority of a variable depends on various factors based on its use in the basic blocks. If a variable is local to a basic block, it has lower priority whereas if it is global and used across many instructions, it is assigned with a higher priority value. GRA marks all the variables as not to be split initially. The following steps are performed for each live interval in the priority queue until all the live intervals are allocated to registers or memory. In each iteration, the

Algorithm 1: Secure Greedy Register Allocation in LLVM

```
Perform a live interval analysis on all variables of $S$;
Mark all the variables of $S$ as no-split;
Calculate the spill weights of all the variables of $S$;
Calculate queue allocation priorities of all variables of $S$;
Enqueue all the variables into queue $Q$ based on priorities;
while $Q$ is not empty do
    Dequeue the variable (let $x$) with highest allocation priority from $Q$ ;
    if $x$ is marked as no-split then
        if machine has free register then
            Assign $x$ to a free register;
        else
            Compare the spill weight of $x$ with spill weights of all interfering variables;
            if Eviction possible then
                Evict the cheaper variable (let $y$) from register (let $r$);
                Assign $x$ to register $r$;
                Enqueue $y$ into $Q$ with same allocation priority;
            else
                Mark $x$ as split;
                Enqueue $x$ into $Q$ with lower allocation priority;
            end
        end
    else
        if Splitting is beneficial then
            Split $x$; Insert split codes for new variables; Calculate the spill weights and
            allocation priorities of all new variables;
            Enqueue all the new variables into $Q$ based on priorities;
        else
            Spill $x$ into memory location $m$;
            Insert spill codes (for memory operations);
            Create new intervals for spills and reloads of $x$;
            if last reload from $m$ then
                Insert code to store zero into $m$;
            end
            Calculate the spill weights and allocation priorities for new intervals of $x$;
            Enqueue the new intervals into $Q$ based on priorities;
        end
    end
end
```

live interval with the highest allocation priority (say x) is de-queued and assigned a register if the machine has a free register (step 9). In case of unavailability of a free register and x is not marked to be split, GRA checks if eviction of an already allocated register (say r) is beneficial (step 13). If yes, it evicts a cheaper interfering interval (say y) from r based on low spill weight and assigns the current interval (x) to the register r. The evicted interval y again en-queued with the same allocation priority. Otherwise, x is marked to be split and en-queued again with lower allocation priority. When there is no free register and x is marked to be split, splitting is performed on x if it is beneficial (step 23). Otherwise, x will be spilled to memory. Splitting divides the live interval x into smaller ones and creates new live ranges. The spill weight is calculated for all new intervals, which are to be enqueued into the priority queue based on allocation priorities. Control goes back to consider the next interval in the queue. When splitting is also not beneficial (step 26), it goes for spilling. Spilling stores the value in memory. It also creates new live ranges after inserting the spill code. The spill weight is calculated for new intervals, and new live intervals are enqueued into the priority queue based on the allocation priorities.

We have analyzed in *Scenario 3* of Section 3.4 that spilling is not relatively secure to the source program. The basic intuition to generate a secure spilling is to remove the secure information stored in the spilled memory locations. Moreover, the secure content must be erased immediately after the last use of it to reduce its lifetime. The specific updates in the greedy register allocation to make spilling secure are as follows:

When a live interval x is chosen for spilling into memory location m, GRA inserts spill instructions and reload instructions after the definition of x and before the use of x, respectively. This requires creating new smaller live intervals for x and deleting the old live interval of x. To use the spilled content, GRA reloads the content from m to some register to perform the desired operations. At this point, we check whether it is the last reload from m (step 30 in Algorithm 1). If it is the last reload and there is no further use of memory content in m, we insert code to store zero into memory location m. This way, we find the last reload of each spilled memory location and insert store zero operations. The last reload of m can be found by live interval analysis.

When a spilled memory location is reused with different sensitive contents, we insert store zero only once after the last reload for the last use of the memory. Therefore, the required number of store zero operations is equal to the unique memory locations used by GRA for spilling purposes. This way, our approach inserts the minimum possible store zero

```
movq %rax, -56(%rbp) //Spill
movq %rbx, -64(%rbp) //Spill
       ...
movq -56(%rbp), %r14 //Reload
       ...
movq -64(%rbp), %r15 //Reload
    ...
movq -64(%rbp), %r16 //Reload
       ...
```

(a)

```
movq %rax, -56(%rbp) //Spill
movq %rbx, -64(%rbp) //Spill
       ...
movq -56(%rbp), %r14 //Reload
movq $0, -56(%rbp) //Spill Zero
       ...
movq -64(%rbp), %r15 //Reload
    ...
movq -64(%rbp), %r16 //Reload
movq $0, -64(%rbp) //Spill Zero
    ...
```

(b)

Fig. 3.8: *(a) Target Assembly Generated in Greedy RA, (b) Target Assembly Generated in Proposed Secure Greedy RA*

instructions into memory. Thus, this is the best possible approach to make spilling secure in register allocation. The overall flow of our proposed secure greedy register allocation (SGRA) approach is shown in Fig. 3.7 where the dotted box demonstrates our proposed spilling approach in GRA.

In Fig. 3.8, we have shown a snippet of target assembly generated by GRA and by our proposed secure greedy register allocation (SecGreedy). Fig.3.8(a) shows that the first spill has a single reload, whereas the second spill has multiple reloads. Our approach inserts spill zero after the last reload of each spilled stack slot, shown in Fig.3.8(b). We have implemented our idea on GRA in LLVM as a proof of concept. It may be noted that the same approach is applicable for all RAs of LLVM to make them secure.

3.6 Experimental Results

3.6.1 Setup

In our experiments, we used an Intel Xeon(R) CPU E5-2620 v4 2.10GHz, 64GB of RAM, running Ubuntu 18.04.3 LTS. We run different benchmarks from the LLVM test suites [8] for all the register allocations in LLVM 10.0.1 [7]. We use the Clang front end of LLVM to generate the LLVM IR. For each benchmark, we generate the four assembly codes corresponding to four register allocations of LLVM, i.e., basic, fast, greedy, and PBQP using the llc tool. Then, we analyze the spill instructions from the generated assemblies. We generate the performance report by llvm-mca and analyze various performance parameters. We automate this entire process to generate all the desired results. We have presented results for 21 benchmarks in Table 3.1 and Table 3.2.

Table 3.1: *Total Spills (#S) and Total Leaks (#L) in Basic, Fast, PBQP, Greedy and Secure Greedy Register Allocations, and Total Registers (#R) in Greedy and Secure greedy Register Allocations.*

Benchmark	Basic		Fast		PBQP		Greedy			SecGreedy		
	#S	#L	#S	#L	#S	#L	#S	#L	#R	#S	#L	#R
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)
almabench	115	15	96	59	115	15	47	15	124	62	0	124
chomp	4	3	116	20	4	3	4	3	118	7	0	118
drop3	1	1	79	28	1	1	1	1	134	2	0	134
exptree	2	2	60	14	2	2	2	2	154	4	0	154
fannkuch	7	4	32	18	7	4	4	4	133	8	0	133
fftbench	24	8	226	44	23	9	16	7	109	23	0	109
five11	5	3	67	10	5	3	3	3	165	6	0	165
fldry	1	1	45	19	1	1	1	1	132	2	0	132
huffbench	3	2	112	50	3	2	3	3	167	6	0	167
linpack-pc	91	22	313	74	93	21	63	17	163	80	0	163
lpbench	8	6	108	22	9	6	9	7	150	16	0	150
matrix	1	1	44	14	1	1	1	1	122	2	0	122
misr	23	10	80	30	19	10	12	10	168	22	0	168
n-body	28	8	58	17	28	8	10	7	139	17	0	139
Oscar	3	2	79	23	3	2	3	3	136	6	0	136
partialsums	30	12	35	25	30	12	22	12	103	34	0	103
puzzle	4	2	33	10	4	2	2	2	148	4	0	148
Queens	8	8	22	14	8	8	8	8	114	16	0	114
queens	1	1	21	9	1	1	1	1	126	2	0	126
recursive	10	5	34	11	10	5	7	5	102	12	0	102
spectral-norm	2	1	36	13	2	1	2	1	138	3	0	138

3.6.2 Results in LLVM

We present the results for total spills and total leaks generated by different register allocations for each benchmark in Table 3.1. We assume all the spilled memory locations contain secure information, which implies each spill is a potential leak. The leaks in Table 3.1 are measured as the number of unique memory locations used for spilling. For the three register allocations *Basic*, *Fast* and *PBQP*, of LLVM, we have presented the results for spills (#S) and leaks (#L) in second to seventh columns, respectively. For the *Greedy* and proposed *SecGreedy* (secure greedy) register allocation, we have presented the results for spills (#S), leaks (#L), and total registers (#R) required in eighth to thirteenth columns, respectively. We measure the total spills from the generated target assembly. It may be noted that actual leaks (#L) are quite low compared to the total spills (#S) since the memory space is heavily reused during spilling. Our method inserts '*L*' store zero operations in secure greedy. The fast RA generates the maximum leaks compared to other RAs. *Fast* RA scans the program linearly at a basic block level. It assigns a register to a variable whenever it appears. It leads to frequent memory operations whenever there is no free register. Therefore, it generates maximum spilling in a program.

Register allocation without spilling is infeasible for real-world applications as the machine has a small number of physical registers (e.g., 32 or 64) compared to the number of variables even in moderate-size programs (typically in the order of hundreds or thousands). This is also confirmed by the results in Table 3.1. Thus, all the register allocations are leaky in LLVM. The leak in the case of proposed secure greedy in column twelve is zero for all the benchmarks since we erase the sensitive content from the spilled memory. The total spills in secure greedy are more than greedy due to the additional insertion of spill zero instructions. We analyzed that there would be no change in the number of registers required at any time for both greedy and secure greedy, as our added instructions do not need any register to perform the memory operations. The total number of registers used by greedy and secure greedy as shown in the tenth and thirteenth columns, respectively in Table 3.1 confirms our observation. Note that the total number of register uses are presented only for these two approaches to show the impact of the greedy approach on our enhancement in secure greedy approach. *As we are storing zero in the memory slots immediately after the last use, the proposed solution is the best possible solution for securing information leaks due to spilling in RAs of LLVM.*

Table 3.2: *Performance Overhead in Greedy Vs Secure Greedy Register Allocation*

Benchmark	Instr.(K)		Cycles(K)		Block RT		Res. Pre.(%)	
	GRA	SGRA	GRA	SGRA	GRA	SGRA	GRA	SGRA
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
almabench	51.8	53.4	44	44.4	202.8	206.8	5.26	7.42
chomp	77.2	79.3	79.6	80.4	332.5	338.3	1.63	1.37
drop3	29.7	29.8	16.8	16.8	99.3	99.5	7.17	4.8
exptree	46	46.5	46	46.5	184.8	185.8	2.38	1.94
fannkuch	15.4	15.8	10.8	11.7	51.0	52.0	3.72	2.57
fftbench	129.8	132.6	137.9	140.4	593.5	607.0	1.67	2.49
five11	29.6	29.9	30.5	30.6	125.8	126.5	0.98	1.3
fldry	32.6	32.8	30.4	30.4	130.3	130.5	4.95	4.95
huffbench	47.6	48.5	36.4	36.7	161.5	164.0	1.38	3.28
linpack-pc	162.4	164.6	101.7	103.7	568.3	573.8	7.27	7.52
lpbench	47.2	47.9	35.6	35.8	168.0	169.8	3.10	2.52
matrix	19	19.1	17.5	17.6	76.3	76.5	0.58	0.59
misr	41.7	42.7	32.7	33.4	149.0	151.5	2.29	4.15
n-body	27.2	27.9	21.6	21.8	96.8	98.5	8.33	7.37
Oscar	35.1	35.4	31.4	31.5	131.5	132.3	0.64	0.64
partialsums	20.4	21.7	17.3	17.9	80.8	84.0	8.14	10.5
puzzle	18.5	18.7	18.1	18.2	75.8	76.3	0.55	1.68
Queens	16.2	17.5	14.7	15.1	60.8	64.0	0.77	4.02
queens	23	23.1	20.3	20.4	92.3	92.5	0.85	0.85
recursive	18.0	18.5	18.7	19.1	81.0	82.3	1.87	2.61
spectral-norm	20.6	20.7	21.1	21.1	82.0	82.3	0.95	0.95
Avg.	**2.09%**		**1.44%**		**1.41%**		**0.43%**	

3.6.3 Performance Overhead

We present the performance overhead of our proposed secure greedy approach (SGRA) over the greedy register allocation (GRA) in Table 3.2. We present four parameters, the number of instructions, the required execution cycles, the block RThroughput (Block RT), and the resource pressure (Res. Pre.), to analyze the overhead. We run each benchmark for 100 iterations in llvm-mca for X86-64 architecture. We show the average overhead for each performance parameter in the last row of Table 3.2. Block RThroughput is the reciprocal of the block throughput. Block throughput is the maximum number of blocks that can be executed per each simulated clock cycle. The maximum number of instructions executed in parallel in each cycle defines the resource pressure. On average, total instructions are increased by 2.09%, cycles by 1.44%, block RThroughput by 1.41%, and resource pressure by 0.43% in SGRA. Also, there is no significant impact on compile time and run time for all the benchmarks. The experimental results confirm that our proposed secure greedy register allocation has a negligible performance overhead.

3.7 Discussion

In this section, we address the following queries which may arise from the proposed solution. *1) Can the attacker read the memory locations anywhere during the execution of the program?*

Although it is theoretically possible to access memory location at a precise time to gather secrets from the memory content, it is practically infeasible for the following reasons. (1) It is extremely hard to identify the exact timing between the register spill and zeroing it (in the order of a few cycles). Even with state-of-the-art side-channel analysis (e.g., using electromagnetic emanation based attack), it is impossible to figure out at the granularity of few instructions. (2) Even if you can time it right, dumping of memory content is not trivial and involves many cycles. (3) Due to these practical considerations, existing approaches [29,64] also assume a similar threat model where an attack is unlikely during the execution of a small function. (4) Note that a program (e.g., online transaction processing) may take millions of cycles, but the attacker needs to figure out exactly when an encryption routine (e.g., AES) is called, which may take 32-64 cycles. In fact, the attacker may not even have 64 cycles since it has to find the opportunity between the spill and the last use when the zeroing effect takes place (instead of the end of the program).

If the attack model assumes that the attacker can dump the memory randomly at the intermediate point, they need to do a large number of iterations of dumping memory content (implying they can dump in almost every cycle) to access the right memory content. This assumption has two fundamental problems:

- One cannot dump during execution through the JTAG port since the I/O speed is significantly slower than the execution speed. One can trace in an internal trace buffer (which can store only a few K bytes, but not the whole cache content) [96].

- If the target is to execute up to a certain point, stop and dump that memory content, that means millions of executions of the programs, which may be practically infeasible. Also, the traditional cache timing attack will not work in this context [93].

Thus, the proposed register allocation approach is secure based on our attack model. The attack model says the attacker has access to the final memory, i.e., at the end of the program. So, the attacker cannot read the memory locations anywhere after the spill and before the last use, as it has no access to memory during the execution of the program.

2) Is the attack model sound and realistic?

The attack model in this thesis is consistent with the threat model in the cybersecurity domain, where the goal of an attacker is to control the system by exploiting the vulnerabilities in software, firmware, or hardware. Secure compilation has received significant attention in recent years to minimize software-level vulnerabilities. Secure compilation targets various avenues, including identification of security goals and attacker models, designing secure languages, devising efficient checking and mitigation techniques, and formal verification of compilation phases. For example, C and C++ do not provide any safety guarantees. Unless the programmer or the compiler takes security into consideration, the lack of memory safety can lead to security vulnerabilities with disastrous consequences. Over the past 12 years, around 70% of all patched security vulnerabilities at Microsoft were memory safety issues [10]. While there are safer languages (e.g., Java, Haskel, and Rust), they are also not immune to low-level attacks.

This thesis focuses on the vulnerabilities introduced during the compilation process. Specifically, this thesis investigates if register allocation introduces memory vulnerabilities that can be exploited by attackers. While our proposed solution is generic across application domains, we would like to illustrate the importance by highlighting cryptographic libraries. Security guarantees provided by cryptographic primitives (e.g., AES for encryption or SHA for hashing) are vital in designing trustworthy software and systems. While formal verification can ensure that the original specification preserves the security guarantees, it is equally important to ensure that the compiler does not introduce any vulnerabilities while generating the target assembly (binary) code. Unless the compilation steps (e.g., register allocation) are trustworthy, we cannot use a potentially vulnerable assembly to design secure and trustworthy systems [16].

3.8 Conclusion

In this chapter, we analyzed the security of register allocation in the presence of splitting and spilling. We found from the experimental results that all the register allocations in LLVM are leaky. We proposed a secure greedy register allocation approach in LLVM and observed that it has negligible performance overhead.

4

QIL: Quantifying Information Leakage for Security Verification of Compiler Optimizations

4.1 Introduction

To verify the relative security between the source and optimized programs, it is first required to quantify the overall information leak in the individual program. In this thesis, we target to measure the information leaks using static taint analysis due to compiler optimizations. Surprisingly, there is no study on the security measures of a program using taint analysis in the context of checking the relative security of compiler optimizations. The exact information leak identification in a program is an undecidable problem, and taint analysis suffers from over-tainting and under-tainting problems due to implicit flows. There are plenty of existing works that target accurate analysis of implicit flow to reduce the over-tainting and under-tainting problems in various analysis contexts. Our work takes motivation from such works for efficient analysis of implicit flows in the context of compiler optimizations.

Threat model: The inputs of a program are either a high or sensitive type and a low or non-sensitive type. We assume the attacker has access to the executable binary of the program. However, he/she has no access to sensitive inputs and their goal is to get these secret inputs. The attacker can execute the program for various inputs and tries to obtain the secret input through the program variables. The same attack model has also been used in Chapters 5 and 6.

To the best of our knowledge, this is the first work that performs a taint analysis to measure the overall information leak in a program considering both explicit and implicit flows due to conditional blocks and loops. The unique contributions of this chapter are as follows:

- Our method introduces a notion of a leak propagation vector for capturing the information leak at various program points.
- It makes some unique analysis of implicit information flow in the context of compiler optimizations.
- It introduces the concept of fixed point of leaks in a loop.
- To identify overall information leakage in a program, we introduce cutpoints for efficient analysis of local taint flow inside conditional blocks and loops.
- Finally, a completely automated method is developed to identify the overall leak in a program. Our method propagates the information leak recursively to the subsequent cutpoints in the program.
- This thesis introduces three quantification parameters of information leakage and uses them for checking the relative security of compiler optimizations.
- Our method shows that well-known compilers like SPARK leak information in their optimization phase.

The rest of the chapter is organized as follows. Section 4.2 presents a motivational example of our work. Section 4.3 presents our program modeling. The quantification of information leakage, the security measurement of a program and quantification parameters for information leakage are presented in Section 4.4, Section 4.5 and Section 4.6, respectively. The formal analysis of our method is given in Section 4.7. The experimental results are shown in Section 4.8. The security analysis of various compiler optimizations using our leak vector is presented in Section 4.9. Finally, Section 4.10 concludes the chapter.

4.2 Motivation

As discussed in Section 2.4.2, taint analysis [37] marks certain inputs as tainted and propagates the taint to variables that are computed from these tainted inputs in an explicit or implicit manner. In explicit information flow, a variable directly depends on a sensitive input[1], whereas in implicit information flow, the value of a variable indirectly depends on a sensitive input. Explicit flows are related to data dependence whereas implicit flows are generally due to control dependence. *A transformed program M_1 is said to be relatively secure if the amount of leaks in M_1 is always the subset of the amount of leaks in the source program M_0.*

```
void p(h, x, y)
{
      //h is high input;
      b = x - y;
      z = x + y;
      if(h > 0)
         a = b - z;
      else
         a = b + c;
}
```

(a) *Source code snippet*

```
void p'(h, x, y)
{
      //h is high input;
      if(h > 0){
        b = x - y;
        z = x + y;
        a = b - z;}
      else{
        b = x - y;
        a = b + c;} }
```

(b) *After code motion*

Fig. 4.1: *Under-tainting and Over-tainting problem in Conditional Speculation*

The taint analysis has two significant limitations: under-tainting and over-tainting. Most of the taint analysis tools do not propagate the taints along control dependencies (also called implicit flows) which leads to an under-tainting problem. This causes false negatives, i.e., there is a leak in the program but the analysis fails to identify it. The following example illustrates this fact.

In Fig. 4.1, the inputs x and y are low type and the input h is the high type. Thus, variables b and z have no leak in Fig. 4.1(a). By code motion, the statement $b = x - y$ is moved to both branches, and the statement $z = x + y$ is moved only to the True branch.

[1] The terms high, tainted, and sensitive are used interchangeably to designate a variable/input/output as sensitive. Similarly, the terms low, untainted, and non-sensitive are used interchangeably to designate a variable/input/output as non-sensitive.

This code motion is correct because b is used in both branches but z is used only in the True branch, and z is never used after the conditional block. As mentioned above, the variable z is not tainted in Fig. 4.1(a). Moreover, z is also not tainted if we ignore the implicit flow of h to z in Fig. 4.1(b). This scenario causes the under-tainting problem since z is leaking information about the high input h due to control flow.

A common approach to overcome the under-taint problem is to identify all the implicit flows in the program and propagate the taints to each such control flow. It means a, b, z will also be marked tained in Fig. 4.1(b). However, this leads to an over-tainting problem, giving false positives, i.e., the tool taints too many variables indiscriminately in the program. In Fig. 4.1(b), for example, b does not leak h in the program because the value of b is the same in both branches. Thus, b is independent of h in the program in Fig. 4.1(b). In reality, there is implicit flow from h only to a and z in Fig. 4.1(b). Thus, we need precise analysis of implicit information flow in a program.

4.2.1 Overview of the Proposed Approach

Finding the exact information leak in a program is an undecidable problem [52, 56]. This thesis tries to quantify the information leakage in the best possible way for the security verification of compiler transformations. To reduce the over-tainting problem, we perform a precise analysis of implicit leaks in a block-wise manner. Specifically, we identify variables inside a "culprit" conditional or loop block that may have different symbolic values in parallel paths inside the block. The primary intuition is that a particular transformation is usually applied to a part of a program by the compiler. Consequently, the taint analysis should be performed "locally or block-by-block" for precise analysis of the implicit leak due to a specific transformation. We, therefore, insert cutpoints in the program to enable "local" analysis of taint flow. We choose cutpoint "intelligently" so that each loop or conditional block can be analyzed locally. We then come up with a taint analysis of a program by propagating leaks across paths. First, we consider a path to find both the explicit leak and implicit leak, then propagate the leak to the subsequent paths. We propagate the leak recursively to cover all the traces in the program. Also, we apply some look-ahead properties to reduce the number of recursions, which in turn reduces the overall complexity of the approach. Finally, our approach measures the overall leak in the program. In a compiler, there would be no correlation between the variables of the source and transformed programs as it adds, removes, and renames the variables while applying various optimizations. Therefore, we do not consider any correlation between the variables of the two programs while checking the relative security of compiler transformations.

To identify the traces, paths, and cutpoints of a program, we need a formal model to represent the program. In this thesis, we represent the program as a finite state machine with datapath (FSMD) [76]. We discuss the modelling of programs as FSMD in the next section.

4.3 FSMD based Modeling of Programs

Let source program M_0 have been optimized into transformed program M_1 after applying a set of compiler optimizations. An FSMD (Finite State Machine with Datapath) is a universal specification model that can represent the data flow and control flow of a program efficiently. The same formulation can also be adapted on other program models like CDFG (Control/Data Flow Graph). The following definitions are adapted from [76].

Definition 4.3.1 (Finite State Machine with Datapath)**.** *An FSMD is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, f, h\rangle$ where*

1. *$Q = \{q_0, q_1, q_2, \cdots, q_n\}$ is the finite set of n control states,*

2. *$q_0 \in Q$ is the reset (initial) state,*

3. *I is the set of primary inputs where $I = I_h \cup I_l$, where I_h is the set of high inputs and I_l is the set of low inputs,*

4. *V is the set of storage variables,*

5. *O is the set of primary outputs,*

6. *$f : Q \times 2^S \rightarrow Q$ is the state transition function,*

7. *$h : Q \times 2^S \rightarrow U$ is the update function of the output and the storage variables, where S and U are defined as follows.*

   (a) $S = L \cup E$ is the set of conditional expressions, where L is the set of Boolean literals of the form b or $\neg\ b$, $b \in L \subseteq I \cup V$ is a Boolean variable and E is the set

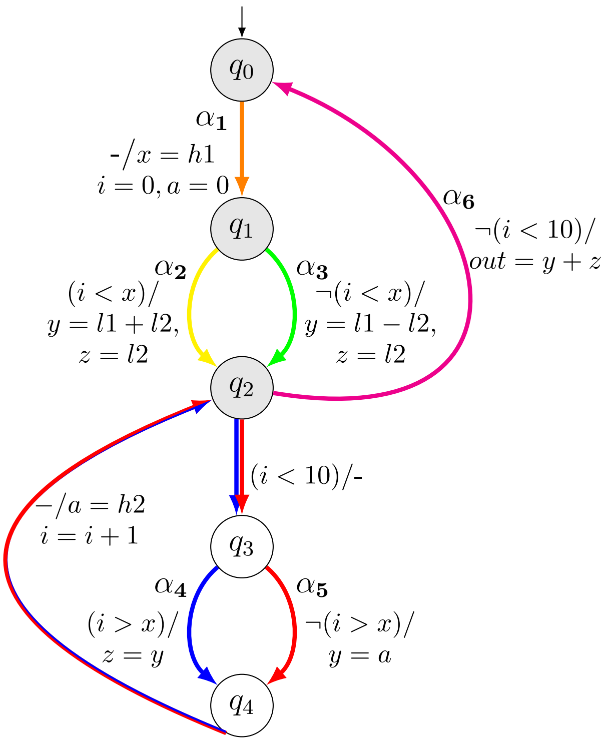
of arithmetic predicates over $I \cup (V - L) \cup Const$ (constants). Any arithmetic predicate is of the form $e1Re2$, where $e1$ and $e2$ are arithmetic expressions and $R \in \{==, \neq, >, \geq, <, \leq\}$.

(b) U is a set of storage or output assignments of the form $\{x = e | x \in O \cup V$, and e is an arithmetic predicate or expression over $I \cup (V - B) \cup Const\}$.

An example code snippet is shown in Fig. 4.2(a), and its corresponding FSMD M is shown in Fig. 4.2(b) for reference. Specifically in M, $Q = \{q_0, q_1, q_2, q_3, q_4\}$, q_0 is reset state, $I = \{l1, l2, h1, h2\}$, $I_l = \{l1, l2\}$, $I_h = \{h1, h2\}$, $V = \{x, y, z, a, i\}$ and $O = \{out\}$. $U = \{x = h1, i = 0, a = 0, y = l1 + l2, \cdots\}$, $S = \{i < x, i < 10, i > x\}$. Some values of f and h are as follows. $f(q0, \{true\}) = q1$, $f(q3, \{i > x\}) = q4$, $h(q1, \{i < x\}) = \{y = l1 + l2, z = l2\}$, $h(q3, \{\neg(i > x)\}) = \{y = a\}$. An FSMD can be automatically extracted from C code [76].

```
void foo(l1,l2,h1,h2)
{   // h1, h2 are high inputs
    x = h1;
    i = 0; a = 0;
    if (i < x)
    {  y = l1 + l2;
       z = l2; }
    else
    {  y = l1 - l2;
       z = l2; }
    while(i < 10)
    {  if(i > x)
         {  z = y; }
       else
         {  y = a; }
       a = h2;
       i = i + 1; }
    out = y + z;}
```

(a)



(b)

Fig. 4.2: *An Example of (a) a Source snippet, (b) Corresponding FSMD M*

4.3.1 Paths and Traces in FSMD

A *path* α, from a state q_i to a state q_j, is a finite sequence of state transitions of the form $\langle q_i \xrightarrow[c_i]{} q_{i+1} \xrightarrow[c_{i+1}]{} \cdots \xrightarrow[c_{i+n-1}]{} q_{i+n} = q_j \rangle$ where $q_k \in Q \; \forall k,\; i \leq k \leq i+n$, and $\exists c_k \in 2^S$ such that $f_k(q_k, c_k) = q_{k+1} \; \forall k,\; i \leq k \leq i+n-1$. The start state q_i and the final state q_j of a path α are denoted as α^s and α^f, respectively. In the FSMD M in Fig. 4.2(b), for example, $\alpha_1 = \langle q_0 \xrightarrow[-]{} q_1 \rangle$, and $\alpha_4 = \langle q_2 \xrightarrow[i<10]{} q_3 \xrightarrow[i>x]{} q_4 \xrightarrow[-]{} q_2 \rangle$ are two paths. In each path of the FSMD, we represent the conditional expression and storage assignments using the notation $s/u1, u2, \cdots$, where $s \in S$ and $u1, u2 \in U$. Note that when s is True, or there is no storage assignments, it is represented as $-$ in the FSMD.

The *condition of execution* R_α of a path α is a logical expression over $I \cup V \cup Const$, which must be satisfied by the initial data state in order to traverse the path α. Thus, R_α is the weakest precondition of the path α [77]. In the FSMD M in Fig. 4.2(b), the R_{α_2} of the path α_2 is $i < x$, similarly, the R_{α_3} of the path α_3 is $\neg(i < x)$.

The *data transformation* S_α of a path α is an ordered tuple $\langle e_j \rangle$ of algebraic expressions over $I \cup V \cup Const$ such that the expression $\langle e_j \rangle$ represents the value of the variable $v_j \in V \cup O$ after execution of the path in terms of the initial data state (i.e., the symbolic values of the variables at the initial state) of the path. In the FSMD M in Fig. 4.2(b), assume the order of variables and output is $\langle x, y, z, a, i, out \rangle$, The S_{α_1} of the path α_1 is $\langle h1, y, z, 0, 0, out \rangle$,the S_{α_2} of the path α_2 is $\langle h1, l1 + l2, l2, 0, 0, out \rangle$. For a path, α, R_α, and S_α can be computed by forward or backward substitution based on symbolic execution [77].

A *computation* or *trace* of an FSMD is a finite sequence of states from the reset state q_0 to itself without having any intermediate occurrence of the reset state. In the FSMD in Fig. 4.2(b), $\tau_0 = \langle q_0 \xrightarrow[-]{} q_1 \xrightarrow[i<x]{} q_2 \xrightarrow[i<10]{} q_3 \rangle$, and $\tau_1 = \langle q_0 \xrightarrow[-]{} q_1 \xrightarrow[i<x]{} q_2 \xrightarrow[i<10]{} q_3 \xrightarrow[i>x]{} q_4 \xrightarrow[-]{} q_2 \xrightarrow[i<10]{} q_3 \xrightarrow[\neg(i>x)]{} q_4 \xrightarrow[-]{} q_2 \xrightarrow[\neg(i<10)]{} q_0 \rangle$ are two traces.

4.3.2 Cutpoints and Path cover

One way of identifying the overall information leak in a program is to measure the leak of each trace in the program and sum up the leaks to measure the overall leak in the program. However, there may be a large or infinite number of traces in a program due to unbounded loops. Therefore, finding all possible traces may not be feasible in practice. For an FSMD M, any trace τ is the concatenation $[\alpha_1 \alpha_2 \cdots \alpha_n]$ of paths of M where $\forall k,\; 1 \leq k < n$, α_k terminates in the start state of the path α_{k+1}, q_0 is both the start state of α_1 and the end state of α_n. For example, the trace τ_1 mentioned above can be represented as $\alpha_1\alpha_2\alpha_4\alpha_5\alpha_6$. A loop may iterate more than one times in a trace. Hence, we have the following definition.

Definition 4.3.2 (Path Cover of an FSMD)**.** *A finite set of paths $P = \{\alpha_1, \alpha_2, \cdots, \alpha_n\}$ is said to be a path cover of an FSMD M if any computation τ of M can be looked upon as a concatenation of paths from P.*

To obtain a path cover, an FSMD is broken down into smaller segments by introducing cutpoints so that each loop in an FSMD is cut in at least one cutpoint. Moreover, there should be no intermediate cutpoint in any conditional block so that each block can be analyzed locally. The set of all paths from one cutpoint to another cutpoint without any intermediary occurrences of cutpoint is a path cover of the FSMD. This follows the Floyd-Hoare method of program verification [62,70]. We denote the path cover of M as P. In this thesis, the cutpoints are defined as follows:

Definition 4.3.3 (Cutpoints in an FSMD)**.** *A state in an FSMD is said to be a cutpoint if it follows one of the following conditions: 1) it is a reset state, 2) it is a loop entry point, or 3) it is a branching state that post-dominates a cutpoint.*

The above-chosen cutpoints indeed will cut each loop in a behaviour. We extract all the post dominators of the FSMD, ignoring the back edges in the FSMD due to loops. If all the paths from a state q_i to the reset state traverse through a state q_j, then we say q_i is post-dominated by q_j or q_j post-dominates q_i. This can be efficiently computed using compiler algorithms [98]. Post dominator ensures that we do not choose any cutpoints inside a conditional block. Our cutpoint selection also ensures that we can analyze a complete nested branching block together and also analyze each loop separately. This will help us to analyze the local impact of code transformations. In the FSMD in Fig. 4.2(b), the states q_0, q_1 and q_2 (in gray) are the cutpoints. The state q_1 is a cutpoint as it is a branching state and post-dominated by cutpoint q_0. The branching state q_3 is not chosen as a cutpoint as it is inside a loop. The path cover for FSMD M is, $P = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}$.

4.4 Quantification of Information Leakage

This section describes the leak in a path and how to propagate the leak over paths and loops in a program. Then, it presents the proposed algorithms for identifying leaks in a program.

4.4.1 Leak Propagation Vector

When there is an information flow from a high input h to a low variable l either in an explicit or implicit manner, we call this a leak of h through l. We define the leak in a path as follows.

Definition 4.4.1 (Leak Propagation Vector)**.** *The leak propagation vector γ_α of a path α in an FSMD is an ordered tuple,*

$$\gamma_\alpha = \langle\langle c\rangle, \langle n_k, n_{k-1}, \cdots, n_1\rangle\rangle \tag{4.1}$$

where $k = |V \cup O|$. The first element c is a x-bit integer (where $x = |I_h|$), represented as $c = \langle b_x, b_{x-1}, \cdots, b_2, b_1\rangle$. In c, $b_j = 1$, $\forall j, 1 \leq j \leq x$, if there is influence of high input h_j in the condition of execution of path α. The second element is a series of integers n_i, $\forall i, 1 \leq i \leq k$, which represents the leak of high inputs through variable v_i either explicitly or implicitly in the path α. Each n_i is a x-bit integer represented as $n_i = \langle b_{i_x}, b_{i_{x-1}}, \cdots, b_{i_2}, b_{i_1}\rangle$ where $b_{i_j} = 1$ if the variable v_i leaks high input h_j; $b_{i_j} = 0$, otherwise.

The leak vector of α, i.e., γ_α, be also referred as the leak γ^{α^f} associated with the final state α^f of the path α. The source snippet in Fig. 4.2(a) has two sensitive/high inputs $h1$ and $h2$, two low inputs $l1$ and $l2$, and six variables $\langle x, y, z, a, i, out\rangle$ in order (including output out). Now, the leak vector for any path α, in FSMD M is $\gamma_\alpha = \langle\langle c_{h_1}c_{h_2}\rangle, \langle x_{h_1}x_{h_2}, y_{h_1}y_{h_2}, z_{h_1}z_{h_2}, a_{h_1}a_{h_2}, i_{h_1}i_{h_2},\ out_{h_1}out_{h_2}\rangle\rangle$. The bit value of x_{h_1} is 1 if x leaks h_1, is 0, otherwise. All the bits are set in the leak vector accordingly.

4.4.2 Explicit Leak in a Path

The explicit leak is due to the data dependencies in a path. In a path α, the explicit leak can be tracked from its data transformations S_α as it represents the symbolic values for each variable and output. Thus, by examining the occurrence of any high input in S_α, all the bits of n_i, for $1 \leq i \leq k$) in the leak vector γ_α for path α is set to either 0 or 1. Similarly, all the bits of c in γ_α of path α is set to either 0 or 1 by examining the occurrence of any high input in the condition of execution R_α. Here, γ_α is independent of any initial leak at the start state α_s, i.e. it considers zero initial leak. For the path α_1 of Fig. 4.2(b), there is an explicit leak of $h1$ through the variable x. There is no leak at the start of the FSMD. So, the explicit leak of path α_1 is $\gamma_{\alpha_1} = \langle\langle 00\rangle, 10, 00, 00, 00, 00, 00\rangle$.

4.4.3 Leak Propagation over Paths

To compute the leak in a trace, the leak of one path needs to be propagated to the successor paths. So, we introduce a concept of leak propagation with initial leak γ^{α^s} associated with the start state α^s of α. Note that γ_α represents the leak of path α with no propagated leak at start state α^s whereas $\gamma_\alpha^{\alpha^s}$ is the leak of path α with propagated leak γ^{α^s} at α^s. The propagated leak $\gamma_\alpha^{\alpha^s}$ of path α with respect to the leak γ^{α^s} at α^s is computed as follows:

$$\begin{aligned} \gamma_\alpha^{\alpha^s}|_v &= \gamma_\alpha|_v,\ \ if\ a\ variable\ v\ is\ defined\ in\ path\ \alpha, \\ &= \gamma^{\alpha^s}|_v,\ \ otherwise. \end{aligned} \tag{4.2}$$

where $\gamma_\alpha|_v$ is the integer n_v in γ_α representing the leak through v in α and $\gamma^{\alpha^s}|_v$ is the corresponding integer for v in γ^{α^s}, i.e., the leak propagated vector at start state of path α. If the variable is defined in the current path, it will overwrite the leak at α^s. Hence, the leak from the current path is considered. If the variable is not defined in the current path, the leak at α^s is propagated to the current path.

The function *FindExplicitLeak()* in Algorithm 2 calculates the explicit leak $\gamma_\alpha^{\alpha^s}$ of a path α with the propagated leak γ^{α^s} at α^s. The function checks for each variable definition in S_α. If it finds a definition, it further checks for the influence of each high input in S_α at line 6 and line 9. The line 6 checks for the explicit presence of a high input in S_α and line 9 checks for the dependency on a variable which is leaky in γ^{α^s} for a high input. Then, it updates the leak vector $\gamma_\alpha^{\alpha^s}$ accordingly for each variable and each high input. However, if it finds there is no influence of high input for a variable v in S_α, it simply propagates the corresponding leak vector value from γ^{α^s} for the variable v in line 14. The first element c in $\gamma_\alpha^{\alpha^s}$ corresponding to R_α is also obtained in a similar manner (lines 16-21) for each high input.

To find the explicit leak of path α_2 in Fig. 4.2(b), the initial leak would be the final leak of path α_1, i.e. $\gamma^{\alpha_1^f} = \gamma^{\alpha_2^s} = \gamma_{\alpha_1}^{\alpha_1^s} = \langle\langle 00\rangle, 10, 00, 00, 00, 00, 00\rangle$. The explicit leak of path α_2 with no initial leak is $\gamma_{\alpha_2} = \phi$. However, $\gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 10\rangle, \langle 10, 00, 00, 00, 00, 00\rangle\rangle$. The variable x leaks $h1$ in α_1, thus, with leak propagation, $h1$ is also leaking explicitly in α_2 followed by α_1. Also, the condition in α_2 is dependent on $h1$, thus c is also updated for $h1$.

Algorithm 2: $FindExplicitLeak(\alpha, \gamma^{\alpha^s})$

```
Input: The path $\alpha$ and the propagated leak $\gamma^{\alpha^s}$ at initial state $\alpha^s$ of path $\alpha$.
Output: Returns $\gamma_\alpha^{\alpha^s}$ the explicit leak of the path $\alpha$.
Compute $R_\alpha, S_\alpha$;
Set each integer in $\gamma_\alpha^{\alpha^s}$ to zero;
foreach high input $h_j \in I_h$ do
    foreach variable $v_i \in V \cup O$ do
        if variable $v_i$ is defined in path $\alpha$, i.e., $S_\alpha|_{v_i} \neq v_i$ then
            if $h_j \in S_\alpha|_{v_i}]$ then
                $\gamma_\alpha^{\alpha^s}|_{(v_i,h_j)} = 1$;
            else
                if $v_i$ is dependent on any variable $x$ in path $\alpha$, (i.e., $x$ is present in
                $S_\alpha|_{v_i}$) and $\gamma^{\alpha^s}|_{(x,h_j)} = 1$ then
                    $\gamma_\alpha^{\alpha^s}|_{(v_i,h_j)} = 1$;
                end
            end
        else
            $\gamma_\alpha^{\alpha^s}|_{(v_i,h_j)} = \gamma^{\alpha^s}|_{(v_i,h_j)}$;
        end
        if $v_i$ occurs in $R_\alpha$ and $\gamma^{\alpha^s}|_{(v_i,h_j)} = 1$ then
            $\gamma_\alpha^{\alpha^s}|_{(c,h_j)} = 1$;
        end
    end
    if $h_j$ occurs in $R_\alpha$ then
        $\gamma_\alpha^{\alpha^s}|_{(c,h_j)} = 1$
    end
end
return $\gamma_\alpha^{\alpha^s}$;
```

4.4.4 Implicit Leak in a Path

The implicit leak of high inputs is due to differences in values for a variable in different control paths. To find out the implicit leaks in a path, we perform a pre-analysis of the FSMD to identify the candidate variables which might be responsible for the indirect leakages due to control flow. The function *PreAnalysis()* is presented in Algorithm 3 for this purpose. Specifically, it finds the distinct parallel paths $\alpha_1, \alpha_2, \cdots, \alpha_n$ between two consecutive cutpoints CP_k and CP_{k+1} and stores into *List*. Then it checks for variables leading to two different expressions by comparing the data transformation S_{α_1} of each variable in $V \cup O$

with the other parallel paths in the *List*. It updates the candidate variable set CV with v at CP_{k+1} upon finding any mismatch of data transformation for the variable v. The data transformation comparison of each variable ensures that, if a variable is not defined in one path but defined in another path, it is also a candidate variable and may lead to an implicit leak. Note that we are not checking the candidate variables in each pair of parallel paths. Instead, we check only with path α_1. If a variable v is not defined in α_1, its symbolic value v is compared with the other parallel paths. This way it reduces the complexity from $O(n^2)$ to $O(n)$ for each variable v where n is the number of distinct parallel paths.

Algorithm 3: $PreAnalysis(M, CP)$

```
Input: The input FSMD $M$ and its set of cutpoints $CP$.
Output: Returns $CV$ and $LCV$, a set of candidate variables and loop candidate
        variables, respectively.
$CV_{cp} = NULL, \forall cp \in CP$;
foreach pair of successive cutpoints $(cp_k, cp_{k+1})$ in $CP$, $1 \leq k \leq |CP|$, where $cp_k$ is a
branching cutpoint and $cp_{k+1}$ post-dominates $cp_k$ do
    Store all the distinct parallel paths $\alpha_1, \alpha_2, \cdots, \alpha_n$ between $cp_k$ and $cp_{k+1}$ in $List$;
    foreach variable $v_i \in V \cup O$ do
        foreach path $\alpha_k \in List - \alpha_1$, where $2 \leq k \leq n$ do
            if $S_{\alpha_1}|_{v_i} \neq S_{\alpha_k}|_{v_i}$ then
                $CV_{cp_{k+1}} = CV_{cp_{k+1}} \cup v_i$;
                break;
            end
        end
    end
end
return $CV$;
```

During the pre-analysis, if a path inside a branch contains a loop, we ignore the loop entry state as a cutpoint. For identifying the candidate variables of a branch, we basically ignore the back edge of the loop inside the branch and consider the parallel paths inside the loop only once. The interleaving of paths inside the loop body will be treated separately (in Section 4.4.5). Further, if a loop condition is influenced by any high input, it also finds the implicit leak of the paths inside the loop.

To find implicit leak, we only consider those branching blocks in which the variables in their conditions are influenced by high inputs. We called those branching blocks as culprit

Algorithm 4: *CheckCulprit*$(\alpha, \gamma^{\alpha^s})$

```
Input: The path $\alpha$ and the leak $\gamma^{\alpha^s}$ at $\alpha^s$.
Output: Returns $TC$ the set of influenced high inputs in path $\alpha$ with leak
        propagation.
$TC = \phi$;
foreach high input $h_j \in I_h$ do
    if $\gamma^{\alpha^s}|_{(c,h_j)} = 1$ then
        $TC = TC \cup h_j$;
    end
end
return $TC$;
```

branches (CB). The function *CheckCulprit()* in Algorithm 4 takes a path α and the leak γ^{α^s} at α^s and returns the set TC of all the high inputs influenced the condition of the path α. Since the sensitivity of a variable in a path depends on the initial leak at the start of the path, we explore the culprit branch by looking into the first element c in $\gamma_{\alpha}^{\alpha^s}$. We update c of the leak vector in Algorithm 2. The Algorithm 4 uses this information to find the culprit branch.

To address the problem of over-tainting due to implicit flows, we propose *"leak propagation rules"* which propagate the leaks in culprit branches. Moreover, our selection of the cutpoint in the FSMD also handles the propagation of implicit leaks. The function *FindImplicitLeak()* in Algorithm 5 takes a path α, the set TC (which stores the high inputs that influenced the condition of execution R_α), and the set CV (which stores the candidate variables impacting implicit leak) as inputs and updates the leak $\gamma_{\alpha}^{\alpha^s}$ in a path α due to implicit flows. It updates the leak vector for each candidate variable in CV_{α^f} at the final state α^f of path α, for each influenced high input in TC. We reduce the complexity of our approach by checking the redefinition of a candidate variable in CV_{α^f}. The function *CheckDef()* (discussed in SubSection 4.5.2) in Algorithm 5 checks for the re-definition of each candidate variable v in all successor paths of α. The function returns True if there exists a re-definition of the variable v in all successor paths. Thus, the leak vector $\gamma_{\alpha}^{\alpha^s}$ is updated when *CheckDef()* returns False. The idea is if the candidate variable is defined in all subsequent future paths, the implicit leak through that variable in the current path can be ignored since the current leak won't have any impact at the end of the execution. Moreover, the variables inside R_α are also leaking the influenced high inputs in TC as R_α

```
Algorithm 5: $FindImplicitLeak(\alpha, TC, CV)$
  Input: The path $\alpha$, $TC$ - the set of influenced high inputs at state $\alpha^s$ and $CV$ - the
         set of candidate variables.
  Output: Returns updated $\gamma_{\alpha}^{\alpha^s}$, with the implicit leak of the path $\alpha$.
  foreach high input $h_j \in TC$ do
      foreach variable $v_i \in CV_{\alpha^f}$ do
          // check for redefinition
          if !CheckDef($v_i, \alpha^f$) then
              $\gamma_{\alpha}^{\alpha^s}|_{(v_i,h_j)} = 1$;
          end
      end
      foreach variable $v_i$ in $R_\alpha$ do
          $\gamma_{\alpha}^{\alpha^s}|_{(v_i,h_j)} = 1$;
      end
  end
  return $\gamma_{\alpha}^{\alpha^s}$;
```

is a single conditional expression. Suppose, loop condition is $i < x$ where i and x both are program variables and x depends on some high input h upon reaching the condition, then i also leads to leakage of h. Thus, function *FindImplicitLeak()* updates the $\gamma_{\alpha}^{\alpha^s}$ for each variable in R_α corresponding to the high inputs in TC, when there is a high input influence for at least one variable in R_α.

The FSMD M in Fig. 4.2(b) has a control block starting at cutpoint q_1. As shown above, the explicit leak of α_2 as $\gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 10\rangle, 10, 00, 00, 00, 00, 00\rangle$. The function *PreAnalysis()* returns the candidate variable set at the final state of the branch as $CV_{q_2} = \{y\}$ due to the different symbolic values of y. Then, *CheckCulprit()* returns the set $TC = \{h_1\}$ due to the presence of x in the condition $i < x$ for this path. Now, *FindImplicitLeak()* updates the implicit leak in α_2 as $\gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 10\rangle, 10, 10, 00, 00, 10, 00\rangle$. It updates the leaks for y since y is in CV_{q_2} and for i since i is in the condition of the path $i < x$ and x is leaking $h1$. Note that variable z has no implicit leak due to the same symbolic value in both branches. Therefore, the updated leak at the final state q_2 (followed by q_1 starting from the reset state q_0) due to both explicit and implicit information flows (with leak propagation) is $\gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 10\rangle, 10, 10, 00, 00, 10, 00\rangle$.

Algorithm 6: $FindLoopLeak(List, p_v, \gamma_{p_v}, \gamma_{loop_v})$

```
Input: List: List of parallel paths inside loop, $p_v$: the path sequence explored so far
       while iterating the loop, $\gamma_{p_v}$: the updated leak for $p_v$, $\gamma_{loop_v}$: the updated leak
       vector of loop.
Output: Returns $\gamma_{loop_c}$, the leak vector of the loop.
foreach path $p \in List$ do
    $p_c = \langle p_v.p \rangle$; i.e., concatenate $p$ with $p_v$;
    $\gamma_{p_c} = FindExplicitLeak(p_c, \gamma_{p_v})$;
    $TC = TC \cup CheckCulprit(p_c, \gamma_{p_c})$;
    $\gamma_{loop_c} = \gamma_{loop_v} \vee \gamma_{p_c}$;
    if $\gamma_{loop_c} = 1$ then
        break;
    else
        if $(\gamma_{p_c} \neq \gamma_{p_v})$ then
            if $(\gamma_{loop_c} \neq \gamma_{loop_v})$ then
                $FindLoopLeak(List, p_c, \gamma_{p_c}, \gamma_{loop_c})$;
            end
        end
    end
end
if $TC \neq \phi$ and $LCV \neq \phi$ then
    $\gamma_{loop_c} = \gamma_{loop_c} \vee FindImplicitLeak(p_c, TC, LCV)$;
end
return $\gamma_{loop_c}$;
```

4.4.5 Leak Propagation over Loops

A loop is visualized as a single path in our algorithm by finding the greatest fixed point of the leak in the loop. In reality, a loop may consist of more than one mutually exclusive path due to conditional branches inside the loop body. In each iteration, however, one of the paths will be executed. The information leakage inside a loop may not be guaranteed by traversing the parallel paths exactly once individually. It is because of the fact that the information flow inside a loop may depend on the interleaving of paths. Therefore, a loop needs to be traversed sufficiently across all its parallel paths to identify the information leakage over the loop. For the loop in Fig. 4.2(b), let the loop first iterates over path α_4, then the high input $h2$ is leaking through low variable a, in the second iteration let it iterates over path α_5, then $h2$ is leaking through low variable y. When the loop iterates again over path α_4, then low variable z is also leaking $h2$. There is no further explicit leak in the loop. This way the greatest fixed point of leakage can be reached for a loop.

When the condition of a loop is tainted, there is a possibility of implicit leak due to the influenced high inputs in the tainted loop condition. This requires a pre-analysis of the loop to find the candidate variables responsible for the implicit leak of the loop. The loop pre-analysis is performed the same way as the branch pre-analysis. It follows the steps in line 4 to 11 of function *PreAnalysis()* in Algorithm 3 to update the set LCV, i.e., the loop candidate variables instead of CV at line 7.

The function *FindLoopLeak()* formalizes the leak propagation over the loop in Algorithm 6. The function *FindLoopLeak()* takes the *List* of parallel paths inside a loop, the path sequence p_v (initialized to ϕ), the leak γ_{p_v} for p_v, and the leak vector of loop γ_{loop_v} explored so far while iterating the loop as inputs and returns the overall leak of the loop γ_{loop_c}, i.e., the greatest fixed point of leak due to the loop. The γ_{loop_c} is initialized to $\langle 0, \phi \rangle)$. A loop may have more than one parallel path. However, in each iteration, it follows a single path. We need to traverse all possible sequences of paths the loop can iterate which leads to information leakage. To identify all these sequences of paths, the algorithm starts with a path p from *List* and concatenates it to the previous path sequence p_v (initialized to ϕ). The new path sequence is now considered as the current path sequence p_c. Then it calls the function *FindExplicitLeak()* to find the propagated leak γ_{p_c} of the current path sequence p_c with initial leak γ_{p_v}. It also calls the function *CheckCulprit()* to check for the influence of any high input on the loop condition and returns the set TC of influenced high inputs. The γ_{loop_c} is now updated with the γ_{loop_v} and the γ_{p_c}, i.e. the leak of the loop explored so far. When it reaches a state such that the loop leads to leak of each high input through each variable, i.e. $\gamma_{loop_c} = 1$, it is no longer required to iterate the loop anymore. Thus, it returns the fixed point of the leak of loop γ_{loop_c}. Otherwise, it checks for a mismatch between γ_{p_c} and γ_{p_v}, also γ_{loop_c} and γ_{loop_v}. If there is mismatch, it calls the function *FindLoopLeak()* recursively and repeats the same process until no new leak is identified. The leak propagated vector γ_{p_c} and γ_{p_v} measures the leak of two consecutive iterations of the loop. However, γ_{loop_c} and γ_{loop_v} measure the overall leak of the loop. Thus, both the parameters must be compared before calling the function *FindLoopLeak()* recursively. Finally, it finds the implicit leak in the loop if the set TC and LCV is not Null, i.e., there exist some candidate variables for the tainted loop.

The state-of-the-art papers find the fixed point of a loop considering a single path inside the loop. However, our approach can find the fixed point of loops with multiple paths inside. The leak propagated vector γ_{loop_c} measures the maximum possible leak in the loop. The maximum leak may not be observable by a single run of the program as the loop condition may be input-dependent. An attacker needs to run the program with different inputs to observe all the leaks in the loop. Finally, we consider the loop as a single path with a single γ_{loop} vector, which indicates the maximum possible information leakages in the loop.

For the loop in Fig. 4.2(b), it has two parallel paths inside it. The α_4 through the True branch from state q_3 and another path α_5 through the False branch from state q_3. For this example the $List$ = $\{\alpha_4,\ \alpha_5\}$, p_v = $NULL$ initially, the propagated leak at entry state q_2 i.e. $\gamma_{p_v} = \gamma^{q_2} = \gamma^{\alpha_2^f} = \gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 10\rangle, 10, 10, 00, 00, 10, 00\rangle$ and the initial loop leak γ_{loop_v} is $\langle 0, \phi\rangle$. Let in the first iteration, α_4 is considered, then the p_c becomes α_4 and $\gamma_{loop_c} = \langle\langle 10\rangle, 10, 10, 00, 01, 10, 00\rangle$ as variable a is leaking h_2. Let α_5 is considered in the next recursive call, then p_c becomes $\alpha_4\alpha_5$, $\gamma_{p_v} = \langle\langle 10\rangle, 10, 10, 00, 01, 10, 00\rangle$ and $\gamma_{loop_c} = \langle\langle 10\rangle, 10, 11, 00, 01, 10, 00\rangle$ as y is leaking $h2$. Let α_4 is considered in the next recursive call, then p_c becomes $\alpha_4\alpha_5\alpha_4$, $\gamma_{p_v} = \langle\langle 10\rangle, 10, 11, 00, 01, 10, 00\rangle$ and $\gamma_{loop_c} = \langle\langle 10\rangle, 10, 11, 01, 01, 10, 00\rangle$ as z is leaking $h2$. There is no more explicit leak due to the loop. The function now checks for the implicit leak (with the propagated leak) and updates the $\gamma_{loop_c} = \langle\langle 10\rangle, 10, 11, 11, 01, 10, 00\rangle$ for variable y, z and i as h_1 is leaked through the variable x in the condition of paths α_4 and α_5. Thus, the final leak of the loop is $\gamma_{loop_c} = \langle\langle 10\rangle, 10, 11, 11, 01, 10, 00\rangle$. Here, we have considered the best path sequence to obtain the fixed point of leak of the loop. In reality, there are other path sequences that will also be traversed by the Algorithm. However, the fixed point of leak for the loop won't change.

4.5 Leak Measurement of a Program

To measure information leakage in a program, we start traversing each path starting from the reset state of the FSMD. We first find all the paths of the FSMD by inserting cutpoints. The function *PreAnalysis()* is called to get the set CV of candidate variables at each cutpoint in CP. Then, the function *FindLeak()* is called to find the overall leak of the FSMD.

Algorithm 7: $FindLeak(M, P, \alpha^s, \gamma^{\alpha^s})$

```
Input: Source FSMD $M$, Path cover $P$ of $M$, Initial state $\alpha^s$, (Initially, $\alpha^s = q_0$),
       and Propagated leak $\gamma^{\alpha^s}$ (Initially, $\gamma^{\alpha^s} = \langle 0, \phi \rangle$).
Output: Returns $\gamma_M$, i.e., the final leak of FSMD $M$
foreach path $\alpha$ emanating from $\alpha^s$ do
    // handle loop leak
    if $\alpha^s = \alpha^f$                                   ▷ loop entry/exit state then
        if ($\alpha^s, \gamma^{\alpha^s}$) exists in LoopLeakList then
            $\gamma_\alpha^{\alpha^s} = \gamma_\alpha^{\alpha^s}$ for pair ($\alpha^s, \gamma^{\alpha^s}$) in LoopLeakList;
        else
            Store all the distinct parallel paths inside the loop emanating from $\alpha_s$ in
            List;
            $\gamma_\alpha^{\alpha^s} = FindLoopLeak(List, \phi, \gamma^{\alpha^s}, \langle 0, \phi \rangle)$;
            Store $\gamma_\alpha^{\alpha^s}$ for the pair ($\alpha^s, \gamma^{\alpha^s}$) in LoopLeakList;
        end
    else
        // handle explicit leak
        $\gamma_\alpha^{\alpha^s} = FindExplicitLeak(\alpha, \gamma^{\alpha^s})$;
        // handle implicit leak
        if $\alpha^s$ is a branching state then
            $TC = CheckCulprit(\alpha, \gamma_\alpha^{\alpha^s})$;
        end
        if $TC \neq \phi$ and $CV_{\alpha^f} \neq \phi$ and $\alpha^f$ post-dominates $\alpha^s$ in $M$ then
            $\gamma_\alpha^{\alpha^s} = \gamma_\alpha^{\alpha^s} \vee FindImplicitLeak(\alpha, TC, CV)$;
        end
    end
    if CheckUsage($\gamma_\alpha^{\alpha^s}, \alpha^f, I_h$) then
        $\gamma_\alpha^{\alpha^s} = FindLeak(M, P, \alpha^f, \gamma_\alpha^{\alpha^s})$;
    else
        $\gamma_M = \gamma_M \vee \gamma_\alpha^{\alpha^s}$;
        // no future use of leaky variables and high inputs
        return $\gamma_M$;
    end
end
```

4.5.1 Algorithm Description

The function *FindLeak()* takes an FSMD M, the path cover P of M, the reset state α^s of M and the initial leak γ^{α^s} (initialized to $\langle 0, \phi \rangle$) of the FSMD M as inputs and computes the overall leak of M in a recursive manner. Specifically, the function identifies the leak in a path α emanating from the current cutpoint (α^s) and traverses the subsequent paths from α^f in a depth first search (DFS) manner to identify the trace level leak in M. The function stops the recursive call when there is no further leak possible. It takes a path α from the α^s and checks whether α^s is loop entry/exit state as we handle loops differently. Moreover, to minimize the re-computation for the fixed point of leakage of loops, we keep track of the already computed leak of loops for each unique the initial leak γ^{α^s} in *LoopLeakList* at each loop entry state α^s. Thus, at each loop entry state α^s, the function *FindLeak()* checks for the existence of a $(\alpha^s, \gamma^{\alpha^s})$ pair in *LoopLeakList* and updates the greatest fixed point of the loop $\gamma_\alpha^{\alpha^s}$ from *LoopLeakList* if exists. Otherwise, it calls the function *FindLoopLeak()* to find the greatest fixed point of the loop with respect to the new propagated leak at α^s and updates the $\gamma_\alpha^{\alpha^s}$. We assume that each loop is executed at least once and it always has a single entry/exit point, i.e., there is no exit, break, return or go to statements inside the loop. Thus, whenever a loop entry state of the FSMD is reached in *FindLeak()*, we find the fixed point of leak in the loop before considering the loop exit path. Specifically, the function considers the fixed point of leak of the loop as the initial leak for the exit path of the loop.

If the path is not a loop, then it calls the function *FindExplicitLeak()* to find the explicit leak of the path α. Then it checks whether the start state α^s is a branching state. If the start state α^s of a path α is a branching state, the function *CheckCulprit()* checks whether the condition of the branch is influenced by any high input. The function *CheckCulprit()* returns the set TC which stores all the high inputs that influenced the condition of the branch. If it finds the path α is a culprit branch leading to implicit leak (i.e., $TC \neq \phi$), and also it has candidate variables (i.e., $CV_{\alpha^f} \neq \phi$), then it calls the function *FindImplicitLeak()* to propagate the implicit leak to the path α. There is a possible scenario that $CV_{\alpha^f} = \phi$ even though $TC \neq \phi$ when α^f is a loop state as we are not storing the candidate variables at loop states inside the branch during pre-analysis. However, the implicit leak for these culprit branches is measured when a direct path (with no loop) inside the branch is traversed during the overall leak measurement. Thus, it does not generate any false negatives.

After successful measurement of leak in a path α, our approach finds the use of the leaky variables before a definition and use of any high inputs in the successor paths in order to reduce the number of recursive calls. This is achieved by the function *CheckUsage()* in line 19 of the Algorithm 7. If it finds any future use of any leaky variable in $\gamma_\alpha^{\alpha^s}$ or any use of high input in I_h, it returns *True* and calls the function *FindLeak()* recursively with α^f as the initial state and $\gamma_{\alpha}^{\alpha^s}$ as the initial leak for the successive path of α. The final leak of the program γ_M is updated with $\gamma_{\alpha}^{\alpha^s}$. In this case, all the traces following the final state α^f at line 22 have the same leak. Finally, the function returns γ_M as the overall program leak. It is possible to obtain the information leak in trace from the γ^{α^s} stored at the cutpoints. Therefore, the trace level leak information can be also calculated by our method in addition to the overall leak of a program. A function call graph for the overall security measurement approach is shown in Fig. 4.3. Note that currently, our approach targets C language. The hierarchical function calls will work with inlining. Our current implementation cannot handle standard libraries, pointers, and dynamic memory allocations. However, our method can be enhanced for other languages and can handle all these scenarios. We leave these as future work.

Consider the FSMD M in Fig. 4.2(b) again to find the overall leak γ_M. To find the overall leak in the program, the function *FindLeak()* starts with path α_1 and finds the explicit leak. Then from α_1^f it traverse either α_2 or α_3. Let α_2 be traversed, followed by α_1. It finds both the explicit and implicit leak of the path α_2 as it is a culprit branch. Then, it traverses the path α_4 or α_5 and finds that it is a loop and thus finds the fixed point of loop leak. After the loop, it follows the exit path α_6 from the loop. It finds the explicit leak of α_6 with the propagated leak at α_6^s as $\gamma^{\alpha_6^s} = \langle\langle 10\rangle, 10, 11, 11, 01, 10, 00\rangle$. Now, $\gamma_{\alpha_6}^{\alpha_6^s} = \langle\langle 10\rangle, 10, 11, 11, 01, 10, 11\rangle$, as the leak from y and z is propagated to *out*. Thus, the final leak of FSMD M is updated as $\gamma_M = \langle\langle 10\rangle, 10, 11, 11, 01, 10, 11\rangle$. There are other recursive calls of the *FindLeak()*, which won't change the final leak of the program. We are not discussing them here for brevity.

4.5.2 Minimizing Complexity by Look Ahead Properties

The function *FindLeak()* calls recursively to measure the overall leak in a program by propagating the leaks. Thus, it starts from the reset state and finds the leak in subsequent paths until it reaches the reset state again (final state). However, these recursive calls may end up with an exponential complexity due to the large number of paths in the program. Therefore, our objective is to use look-ahead properties to minimize the recursive calls. The function *CheckUsage()* at line 19 in Algorithm 7 is used to minimize the recursive calls of the overall security measurement method. Before each recursive call to function *FindLeak()*,

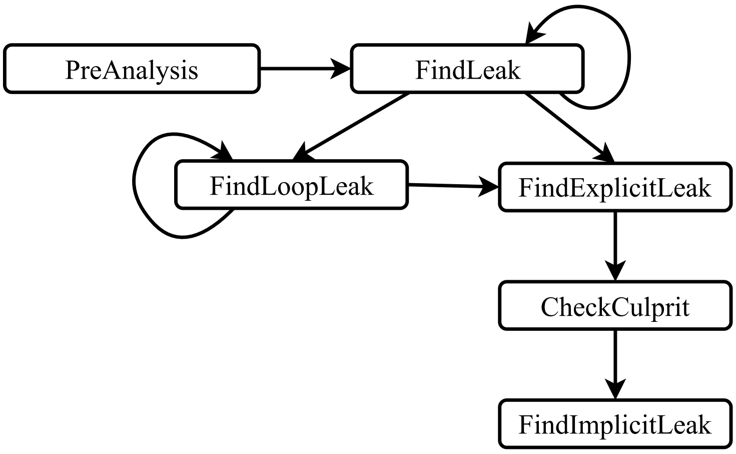
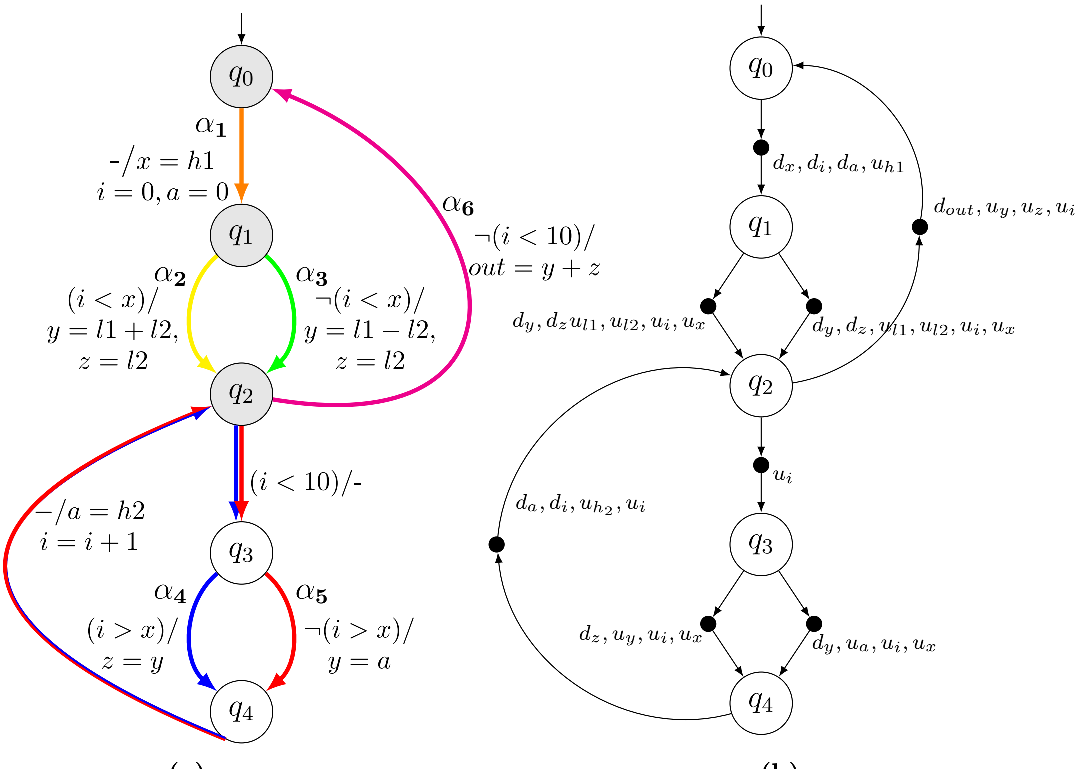


Fig. 4.3: *Function call graph for Quantification of Information Leak*

the *CheckUsage()* function checks the impact of the current leak in the future paths of the FSMD M. Then, it takes a decision on the propagation of the current leak to the subsequent paths, i.e., the recursive call to the function *FindLeak()*. The impact of the current leak on future paths depends on the use of the current leaky variables without a redefinition and the existence of the use of any high inputs in any subsequent paths of M. If these possibilities hold at a cutpoint, it signifies we need to propagate the current leaks to subsequent paths for identifying further leaks in the program. Otherwise, the recursive call is stopped and the current leak measured so far is the overall leak of the program. Thus, the function *CheckUsage()* is vital for reducing the overall complexity of our proposed method.

This function uses specific Computation Tree Logic (CTL) properties on an equivalent Kripke structure [46] of an FSMD. The model checking of a property involves three propositions, d_v and u_v, for each variable v in V, and u_h, for each h in I_h, where d_v, u_v, and u_h represent $defined(v)$, *used*(v) and *used*(h), respectively. The Kripke structure has a state for each state of the FSMD. Also, there is a dummy state added for each transition of the FSMD. The Kripke structure is shown in Fig. 4.4(b) for the FSMD in Fig. 4.4(a). The dummy states are represented as black circles. In a state, the proposition d_v is true when a variable v is defined by some operation in the corresponding transition in the FSMD. Similarly, u_v is true when a variable v is used by some operation in the corresponding FSMD. In a state, if any proposition is not present, then the negation of the proposition is true in that state. Any CTL model checker like NuSMv [45] can verify a temporal property on this Kripke structure at any state. Note that the Kripke structure design avoids the state



(a) **(b)**

Fig. 4.4: *Kripke representation: (a) Corresponding FSMD M for Source snippet in Fig. 4.2(a), (b) Kripke structure obtained from FSMD M*

explosion problem by using the Boolean versions for programs variable over Integers and real numbers. It is only required to store two Boolean values for each variable at each state of the Kripke model.

The function *CheckUsage()* at line 19 in Algorithm 7 checks "v is used before definition" and "h is used". This is achieved by applying the CTL formulae $E[(\neg d_v)\ U\ u_v]$ and $EF(u_h)$ at the final state α^f for each leaky variable v and each high input h in the leak vector $\gamma_\alpha^{\alpha^s}$. Here, E and U represent the existence and until in CTL. The model checker returns True for CTL formulae if it finds the use of variable v before a definition of v in any successor path or the use of any high input h in any successor path.

We have also applied look-ahead properties to avoid the implicit flows to the subsequent paths in the FSMD M. The functions *CheckDef()* at line 3 in Algorithms 5 is used for that purpose, which further reduces the over-approximation of information leaks. This function checks "v is defined always". Our approach updates the γ_α for those candidate variables which do not have a redefinition in all successor paths, i.e. when *CheckDef()* returns false. This is achieved by applying the CTL formula $AF(d_v)$ at the final state α^f, where AF represents like prefuture states in all paths. The CTL formula is true only if there is no re-definition of variable v in all successor paths.

4.6 Quantifying Parameters for Information Leakage

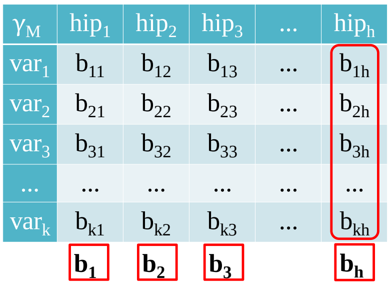
In this thesis, we present three parameters for quantifying the information leakage in an FSMD M. These quantification parameters are directly derived from the overall leak measured in the leak propagation vector γ_M for the program M. Let us consider the leak vector γ_M in Figs. 4.5 to 4.7 with k variables (var) and h high inputs (hip) where b_{ij} represents the leak of high input hip_j through variable var_i such that $1 \leq i \leq k$ and $1 \leq j \leq h$.

4.6.1 Quantification Approaches for a Program

For $\gamma = \langle\langle c\rangle, \langle n_k, \cdots, n_1, n_0\rangle\rangle$ where $n_i = (b_{i_h} \cdots b_{i_1} b_{i_0})$, $1 \leq i \leq (V \cup O)$, i.e., $k = V \cup O$, we propose the three quantifying parameters as follows.

Parameter 1: We find the total number of high inputs leaked in a program irrespective of through how many low/local variables. A high input leaking through more than one low/local variable is considered as one leak. The following equation measures $leaky_h$, the number of leaky high inputs.

$$leaky_h = \sum_{j=1}^{h} \left(\bigvee_{i=1}^{k} b_{i_j} \right) \tag{4.3}$$



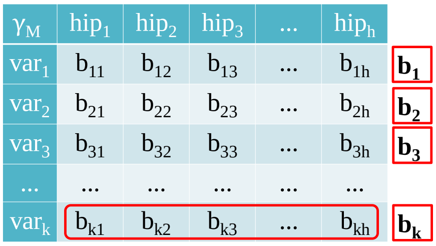
γ_M	hip_1	hip_2	hip_3	...	hip_h
var_1	b_{11}	b_{12}	b_{13}	...	b_{1h}
var_2	b_{21}	b_{22}	b_{23}	...	b_{2h}
var_3	b_{31}	b_{32}	b_{33}	...	b_{3h}
...	...	...	...	...	...
var_k	b_{k1}	b_{k2}	b_{k3}	...	b_{kh}
	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{b_3}$		$\mathbf{b_h}$

Fig. 4.5: *Measuring the quantification parameter 1 using the leak vector*

If we make a union of each column of γ_M we get a boolean bit b_j for $1 \leq j \leq h$, then if we sum up all these bits, we get the parameter 1. This illustration is shown in Fig. 4.5.

Parameter 2: We find the total number of unique leaky variables in a program with respect to high inputs leaked. A variable leaking more than one high input is considered as one leak. The following equation measures $leaky_{uv}$, the number of unique leaky variables.

$$leaky_{uv} = \sum_{i=1}^{k} \left(\bigvee_{j=1}^{h} b_{i_j} \right) \tag{4.4}$$



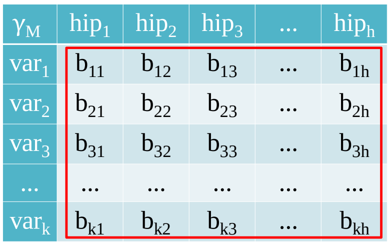
γ_M	hip_1	hip_2	hip_3	...	hip_h	
var_1	b_{11}	b_{12}	b_{13}	...	b_{1h}	$\mathbf{b_1}$
var_2	b_{21}	b_{22}	b_{23}	...	b_{2h}	$\mathbf{b_2}$
var_3	b_{31}	b_{32}	b_{33}	...	b_{3h}	$\mathbf{b_3}$
...	...	...	...	...	...	
var_k	b_{k1}	b_{k2}	b_{k3}	...	b_{kh}	$\mathbf{b_k}$

Fig. 4.6: *Measuring the quantification parameter 2 using the leak vector*

If we make a union of each row of γ_M we get a boolean bit b_i for $1 \leq i \leq k$, then if we sum up all these bits, we get the parameter 2. This illustration is shown in Fig. 4.6.

Parameter 3: We find the total number of leaky variables in a program with respect to unique high input. A variable leaking two high inputs is considered as two leaks, one for each high input. Similarly, a high input leaked by two variables is considered as two leaks for that high input. The following equation measures $leaky_v$, the number of leaky variables.

$$leaky_v = \sum_{j=1}^{h} \left(\sum_{i=1}^{k} b_{i_j} \right) \tag{4.5}$$



γ_M	hip_1	hip_2	hip_3	...	hip_h
var_1	b_{11}	b_{12}	b_{13}	...	b_{1h}
var_2	b_{21}	b_{22}	b_{23}	...	b_{2h}
var_3	b_{31}	b_{32}	b_{33}	...	b_{3h}
...	...	...	...	...	...
var_k	b_{k1}	b_{k2}	b_{k3}	...	b_{kh}

Fig. 4.7: *Measuring the quantification parameter 3 using the leak vector*

If we sum up each boolean bit b_{ij} of γ_M for $1 \leq i \leq k$ and $1 \leq j \leq h$, we get the parameter 3. This illustration is shown in Fig. 4.7.

The proposed parameters are used to verify the relative security between the source and transformed programs, i.e., discussed in the next subsection.

4.6.2 Quantification Approaches for Relative Security

The FSMD representation of source FSMD M_0 and the transformed FSMD M_1 are $\langle Q_0, q_{00}, I, V_0, O, f_0, h_0\rangle$ and $\langle Q_1, q_{10}, I, V_1, O, f_1, h_1\rangle$ respectively. The leak vector for both FSMD M_0 and M_1 is represented as $\gamma_0 = \langle\langle c_0\rangle, \langle n_{0v_0}, \cdots, n_{01}, n_{00}\rangle\rangle$ and $\gamma_1 = \langle\langle c_1\rangle, \langle n_{1v_1}, \cdots, n_{11}, n_{10}\rangle\rangle$, respectively where $n_{0i} = (b_{0i_h} \cdots b_{0i_1} b_{0i_0})$, $1 \leq i \leq V_0 \cup O$ and $n_{1i} = (b_{1j_h} \cdots b_{1j_1} b_{1j_0})$, $1 \leq i \leq V_1 \cup O$, where, h is the number of high inputs, i.e., $h = |I_h|$, V_0 and V_1 are the set of program variables in FSMD M_0 and FSMD M_1, respectively, I and O are the set of inputs and outputs for both M_0 and M_1, respectively. It may be noted that the input and output sets are the same for both M_0 and M_1 because M_1 is the transformed version obtained from M_0. Here, in γ_0 and γ_1, $v_0 = |V_0 \cup O|$ and $v_1 = |V_1 \cup O|$, b_{si_j} and b_{ti_j} are the corresponding leak values in n_{si} and n_{ti} for i^{th} variable and j^{th} high input, respectively.

Definition 4.6.1 (Secure Program Transformation). *A program transformation is said to be secure if there is no leak in the transformed FSMD. Formally, M_0 and M_1 are secure, denoted by $M_1 \simeq_S M_0$ (read as M_1 is securely equivalent to M_0), if $\gamma_0 = \langle 0, \phi\rangle$ and also, $\gamma_1 = \langle 0, \phi\rangle$, i.e., there is no information leakages in the respective FSMDs.*

Definition 4.6.2 (Relatively Secure Program Transformation). *Two FSMDs are said to be relatively secure, denoted by $M_1 \simeq_R M_0$ (read as M_1 is relatively secure to M_0), if there is a leak in M_1, there must be a corresponding leak exists in M_0. Formally, M_0 and M_1 are said to be relatively secure if they follow both the necessary condition followed by the sufficient condition. The necessary condition is as follows:*

$$\left(\bigvee_{i=1}^{V_1} b_{1i_j} - \bigvee_{i=1}^{V_0} b_{0i_j}\right) \leq 0,\ \forall j \in h \tag{4.6}$$

The above equation restricts any new leak of high input in M_1, which is not there in M_0. Also, it can verify whether M_1 is more secure than M_0. The sufficient condition is as follows:

$$\left(\sum_{i=1}^{V_1} b_{1i_j} - \sum_{i=1}^{V_0} b_{0i_j}\right) \leq 0, \forall j \in h, \tag{4.7}$$

The Eq. (4.6) satisfies when a high input leaking in M_1 is also leaking in M_0 irrespective of through how many low variables. The Eq. (4.7) satisfies when, for all high inputs, the number of variables leaking a specific high input in M_1 is not more than that of in M_0. Thus, the sufficient condition needs to be checked only when the necessary condition holds.

For both FSMDs M_0 and M_1, the necessary condition for relative security in Eq. (4.6) checks for the Parameter 1 in Eq. (4.3) and the sufficient condition in Eq. (4.7) checks for the Parameter 3 in Eq. (4.5) for each high input.

4.7 Correctness and Complexity

Detecting the information leakage in a program is undecidable in general [52, 56]. Thus, achieving completeness for ensuring security properties in a program is impossible. Therefore, we aim to develop a sound approach. We ensure it does not result in any false positives.

4.7.1 Soundness and Termination

Our security measurement approach ensures that there is leakage in a program when the final leak vector γ_M obtained by our method for an FSMD is non-zero. We update the leak vector in three possible scenarios: i) when there is an explicit leak due to direct dependencies on high inputs, ii) when there is an implicit leak due to the control flow, and iii) while computing the fixed point of leaks (both explicit and implicit leak) inside a loop. The non zero of γ_M ensures either there is an explicit flow of leak to a program variable or an implicit flow of leak (i.e., there exist at least two traces in the program for which the attacker can get some information about the sensitive information). Therefore, our method does not give any false positive results. Thus, our security measurement approach is sound.

Theorem 4.7.1. *The Algorithm 7 always terminates.*

Proof. The function *FindLeak()* in Algorithm 7 finds all the traces in an FSMD to find the overall FSMD leak in the worst-case scenario. However, an FSMD has a finite number of traces as we consider each loop as a single path. Now, we need to prove the function *FindLoopLeak()* always terminates. The function *FindLoopLeak()* calls itself recursively upon satisfying two conditions, i.e., when the leak vector of two consecutive iterations of the loop (γ_{p_v} and γ_{p_c}) is mismatched and the overall leak vector of the loop (γ_{loop_c}) for two consecutive iterations of the loop is also mismatched. Thus, in the worst case, the loop iterates until each bit of γ_{loop_c} is 1. However, γ_{loop_c} has a finite length. Thus, it calls itself recursively for a finite time. Therefore, Algorithm 7 always terminates. □

4.7.2 Complexity Analysis

The complexity of the overall security measurement approach depends on the following factors: i) the complexity of finding the leak for a given path. ii) the number of times *FindLeak()* is called. The first factor depends on the complexity of finding the loop leak if the given path is a loop. Otherwise, it is the complexity of finding an explicit leak followed by the implicit leak after checking for the branch culprit. For finding the implicit leak, preanalysis of the FSMD is done before calling the function *FindLeak()*.

Assume there are C cutpoints, maximum k_1 parallel paths between two consecutive cutpoints, H high inputs, and V variables (including the outputs) in an FSMD. The function *PreAnalysis()* takes $O(C \cdot V \cdot k_1)$ time in worst case. The function *FindExplicitLeak()* in worst case scenario takes $O(H{\cdot}V)$ time. The function *CheckCulprit()* and *FindImplicitLeak()* takes $O(H)$ and $O(H \cdot V)$ time in worst case. We consider the function *CheckDef()* takes a negligible amount of time. Assume there are k_2 parallel paths inside a loop. The worst case time complexity for one iteration of the loop would be the addition of complexity of pre-analysis, explicit leak, checking culprit, and implicit leak, i.e., $(O(V \cdot k_2) + O(H \cdot V + H + H \cdot V) \simeq O(H \cdot V))$. So, for k_2^V iterations of the loop, the worst case complexity would be $O(k_2^V \cdot H \cdot V)$. However, generally, a loop has a few paths (i.e. the value of k_2 is a small integer constant). Therefore, for the first factor, i.e., to find the leak of a given path, the complexity would be $O(C \cdot V \cdot k_1 + k_2^V \cdot H \cdot V)$.

Now for the second factor, the function *FindLeak()* calls itself recursively at each cutpoint for each path emanating from a cutpoint. Thus, it calls for $(k_1 + k_1^2 + \cdots + k_1^{C-1})$ times which is of the order of $O(k_1^C)$. Therefore, the complexity of the overall security measurement approach is $O(k_1^C(C \cdot V \cdot k_1 + k_2^V \cdot H \cdot V)) \simeq O(k^{(C+V)} \cdot H \cdot V)$ in the worst case if we consider the $Max(k_1, k_2)$ as k.

In the best-case scenario, there is no recursive call for the function *FindLeak()*, and there is a single path inside a loop. Thus, the best-case complexity would be the product of time complexity for a given path and the maximum-paths between the two consecutive cutpoints starting from the reset state. Therefore, the best case complexity is $O(k \cdot H \cdot V)$.

4.8 Experimental Results

4.8.1 Setup

The proposed security measurement approach has been implemented in C [1]. We ran a variety of benchmarks for various compiler optimizations in the SPARK [68] high-level synthesis (HLS) tool. This tool applies a wide range of optimizations on the input C code for efficient synthesis and provides the optimized C code as a by-product. The applied optimizations include code motion, common sub-expression elimination, dead code elimination, copy and constant propagation, and many more. We generate the original or source FSMD, i.e., M_0, from the input C and the transformed or optimized FSMD, i.e., M_1, from the optimized C code for each benchmark. The process of generating the FSMDs from C code has been automated in our tool. The overall tool flow is shown in Fig. 4.8. Note that our approach can be adapted for any general-purpose compilers. We have used an Intel Xeon(R) CPU E5-2620 v4 2.10GHz, 64GB of RAM, running Ubuntu 18.04.3 LTS in our experiments.

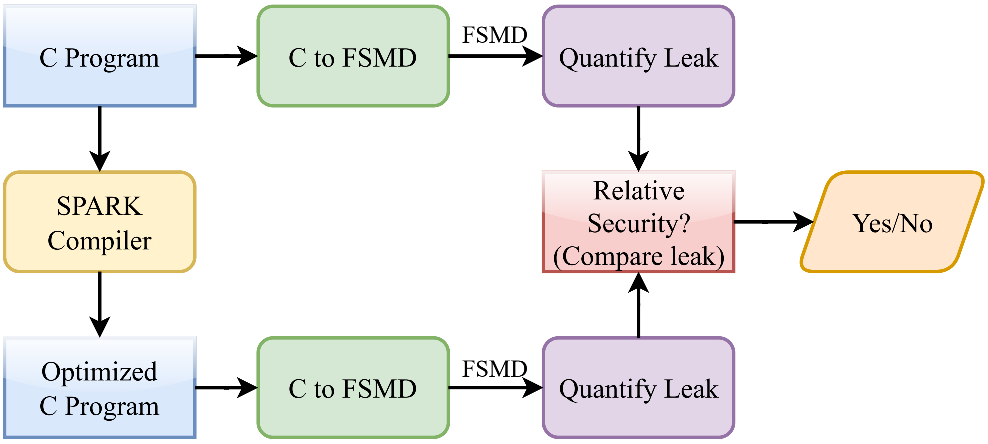


Fig. 4.8: *Overall flow for relative security of information leakage*

4.8.2 Performance Measures

We have presented the results for 11 benchmarks (taken from [22]) in Table 4.1. The number of cutpoints (#CP) and paths (#P) are presented in Columns 2 and 3, respectively, for each benchmark. It may be noted that SPARK does not alter the control structure during

[1] Available at https://github.com/PriyankaPanigrahi/QuantifyLeak

Table 4.1: *Performance Measures for Benchmarks*

Bench	#CP	#P	#IN	#H	#Explicit		#Implicit		RS?	ET (Sec)	
					M_0	M_1	M_0	M_1		M_0	M_1
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
dct	1	1	8	2	54	64	0	0	No	0.002	0.003
				2	46	56	0	0	No		
				2	56	64	0	0	No		
				2	48	52	0	0	No		
modn	2	10	1	1	5	6	3	3	No	0.003	0.004
diffeq	2	3	2	1	13	17	1	2	No	0.001	0.001
perfect	3	6	1	1	1	1	2	3	No	0.002	0.002
barcode	2	65	4	2	0	0	20	22	No	0.023	0.02
parker	3	9	6	2	9	13	4	11	No	0.003	0.008
				2	7	8	7	15	No		
				3	9	13	11	22	No		
find min8	8	15	8	2	6	8	0	0	No	0.028	0.024
				1	3	1	0	0	Yes		
waka	2	4	20	2	7	7	8	10	No	0.002	0.002
				2	15	15	0	0	Yes		
				4	15	15	16	18	No		
				4	17	17	8	9	No		
lru	3	101	2	2	0	0	32	32	Yes	0.036	0.037
qrs	13	56	1	1	44	35	12	12	Yes	51.55	22.18
ieee754	6	519	2	2	32	32	12	10	Yes	131.29	108.5

optimization. Hence, the number of paths is the same in both FSMDs M_0 and M_1. A trace is a concatenation of paths. Thus, the number of traces is quite large for the benchmarks. The number of inputs for each benchmark is presented in Column 4 (#IN). We select a random subset of inputs as high inputs for each benchmark. In column 5 (#H), we present the number of high inputs chosen. For some benchmarks like *parker, findmin8, dct,* and *waka*, we consider different combinations of high inputs. Note that the same number of high inputs for a benchmark actually represents a different set of high inputs with the same count. For example, *parker* has three rows (2, 2, 3) in Column 5. Although the first two rows have two high inputs, they are actually different inputs. We found that all the benchmarks lead to information leakage of high inputs. Our quantification parameter $leaky_h$ (i.e. number of high input leaked) confirms the same.

The total number of leaky variables $\#leaky_v$ is the combination of both explicit flows and implicit flows. We have presented the total number of explicit flows and implicit flows detected by our approach in Columns 6-9 of Table 4.1 for both M_0 and M_1. Its clear that there are explicit leaks in both M_0 and M_1 in almost all cases. The explicit leaks are not the same in M_0 and M_1 in 12 out of 20 cases. We found that there is no implicit flow for both *findmin8* and *dct*. As identified, in 9 out of 11 benchmarks, there is implicit information flow, and the implicit flow in M_0 and M_1 is not the same for six benchmarks.

Finally, we have shown the average execution time (ET) required for each benchmark in the last two columns of Table 4.1. The execution time depends on three parameters: the number of cutpoints, the number of parallel paths, and how many times the function *FindLeak()* is called recursively. We found that the execution times for all cases in *parker, findmin8, dct* and *waka* are almost the same. We, therefore, present the average execution time of one of all runs for each benchmark. The benchmarks *qrs* and *ieee754* take more time due to their large number of paths and cutpoints.

4.8.3 Results on Quantification Parameters

The first quantification parameter, $leaky_h$ is the same for both M_0 and M_1 in most of our experiments. This shows that identifying only the number of leaky high inputs is necessary but not sufficient to ensure the relative security of compiler transformations. This justifies the sufficient condition of our definition of relative security (Definition 4.6.2).

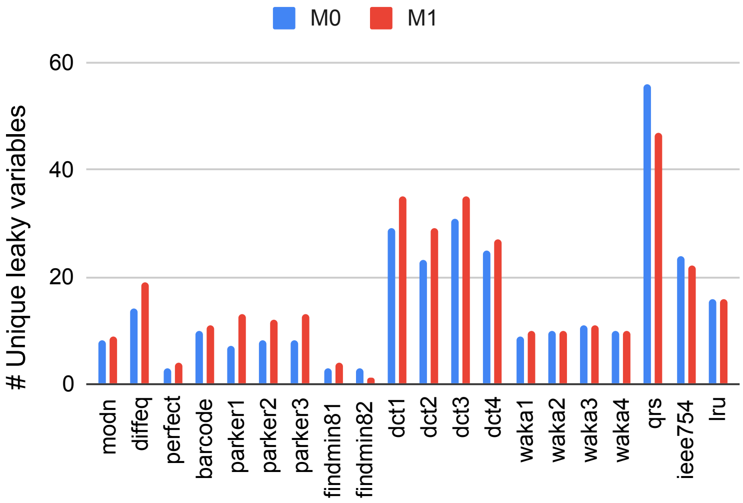


Fig. 4.9: *Unique Leaky Variables (#$leaky_{uv}$) in M_0 Vs M_1*

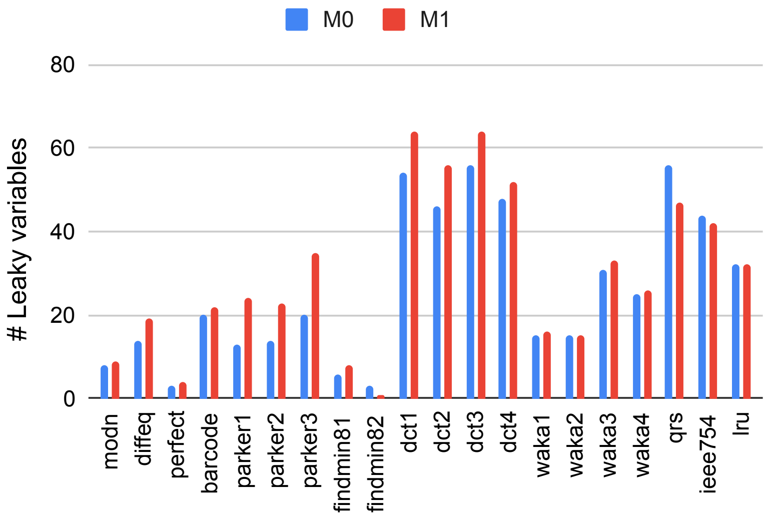


Fig. 4.10: *Leaky Variables (#leaky$_v$) in M_0 Vs M_1*

Our second and third quantification parameters for total number of unique leaky variables (*#leaky$_{uv}$*) and total number of leaky variables (*#leaky$_v$*) concerning high inputs are presented in Fig. 4.9 and Fig. 4.10, respectively. Benchmarks with different high input sets are named as *parker1, parker2, parker3* and so on. It's obvious that when the #high is 1, the *#leaky$_{uv}$* is the same as the *#leaky$_v$*. The *#leaky$_{uv}$* and *#leaky$_v$* results for the first three benchmarks and *qrs* reflect the same as they have one high input. It is the same for *lru* and, in some cases for *waka* and is less in M_0 for *qrs, ieee754* and one case in *findmin8*. Similar results have been obtained for *#leaky$_v$* as well. This shows that the applied optimizations by SPARK either do not impact the information leakage of the source programs or reduce the information leakage in the optimized code for these cases. Importantly, we found that for M_1, the *#leaky$_{uv}$* is more than that of M_0 for 13 out of the 20 cases. The *#leaky$_v$* value for M_1 is more than that of M_0 in 15 out of the total 20 scenarios. This proves that SPARK introduces new information leaks during optimizations in most of the scenarios.

4.8.4 Results on Relative Security

We also check the relative security (RS?) between M_0 and M_1 using our proposed quantifying parameters in Eq. (4.6) and (4.7), and presented the result in Column 10. Out of a total of 20 cases, only five are relatively secure, and for 15 cases, M_1 is not relatively secure to

M_0. We have presented the overall flow for verifying the relative security between the source and transformed programs in Fig. 4.8. Our quantification tool takes both the source and optimized FSMDs individually and measures the overall leak in both programs. Then, the overall leaks are compared to verify the relative security. However, this approach cannot identify the exact point of information leak in the program. This issue has been resolved in our translation validation method (TVIL) in the next chapter.

4.8.5 Scalability of Proposed Approach

We have shown the results for the lines of code (LoC) and the average execution time (in sec) in Fig. 4.11 for our benchmarks. The graph shows the runtime of our tool is growing almost linearly with the code size. This implies our method is scalable enough to handle small to moderate size programs.

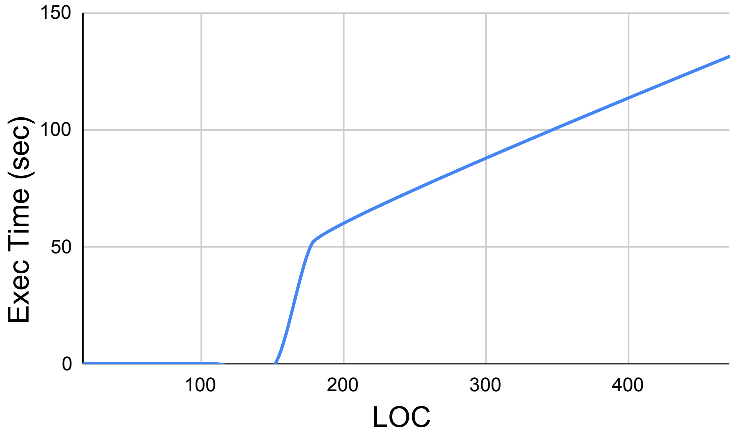


Fig. 4.11: *LoC Vs Execution Time (sec)*

4.8.6 Comparison with Existing Approaches

We compare total leakage ($\#leaky_v$) with the under-tainting and over-tainting results and found that our approach successfully detected the false negatives due to under-tainting and also ignored the false positives caused by over-tainting. We have shown these results in Fig. 4.12 for the benchmarks for which we found the difference in leaks. The overall result confirms that compiler optimization in SPARK introduces security vulnerability in the optimized code for moderate-sized programs. Our quantification parameters can identify such information leakage in the source and optimized programs, which helps us to check the relative security between them.

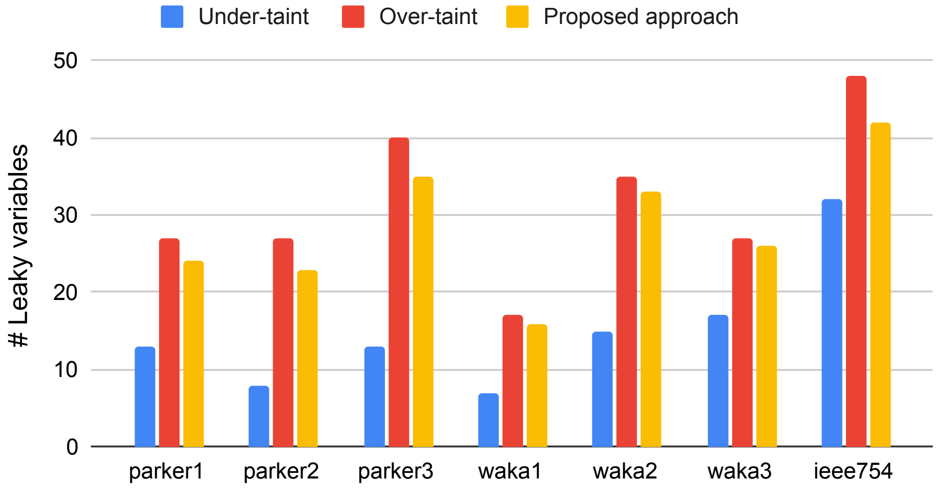


Fig. 4.12: *Comparisons with other Taint Approaches*

4.9 Security Analysis of Various Compiler Optimizations

The compiler applies various optimizations. In this section, we analyze the security of such optimizations using our proposed quantification method. Based on our analysis, we have categorized the optimizations into two classes: (i) secure and (ii) insecure optimizations. We discuss the insecure optimizations followed by the secure optimizations with examples in the next two subsections.

Let γ_S and γ_T be the leak vector of the program before transformation (Source Program S) and after transformation (Optimized Program T), respectively. The V_0 and V_1 are the set of variables in S and T, respectively. The I_h is the set of high inputs. For ease of analysis, here we consider an array as tainted if one of the array elements becomes tainted. Thus, we handle the array just like a variable in our leak vector. However, our approach can be easily extended to handle element-level security analysis of arrays by representing the data transformation and condition of execution of a path as done in [23].

4.9.1 Insecure Compiler Optimizations

We found that some of the common compiler optimizations are actually leaky. We discussed them with examples here.

4.9.1.1 Dead Store Elimination

The compiler applies dead store elimination (DSE) to improve the performance of the code by removing the operations that do not impact the output. An instance of DSE is shown in Fig. 4.13, where the dead store 'x=0' is eliminated after the compiler applies optimization. The transformation is functionally correct since the output won't be impacted due to this transformation. However, it leads to the leak of secret high input $h1$ through variable 'x' after transformation in Fig. 4.13(b). Our leak vectors are $\gamma_S = \langle\langle c\rangle, \langle x_h\rangle\rangle = \langle\langle\phi\rangle, \langle 0\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle x_h\rangle\rangle = \langle\langle\phi\rangle, \langle 1\rangle\rangle$. The necessary condition for relative security in Definition4.6.2 says if high input $h1$ is leaking in M_1, it must be leaking in M_0. Here, $k1 = k2 = 1$ and $h1 = 1$, then Eq. (4.6) expands into $(1-0) \not\leq 0$. Therefore, DSE is not relatively secure.

```
void foo()
{
  x = h1; //use x;
  x=0; //dead store
}
```

(a)

```
void foo()
{
  x = h1;
  // use x;
}
```

(b)

Fig. 4.13: *An Example of Dead Store Elimination*

4.9.1.2 Single Static Assignment

The single static assignment (SSA) helps in improving the register usage by splitting the lifetime of a variable such that it can be mapped to multiple registers. In Fig. 4.14, variable 'x' is converted to SSA. The high input $h1$ stored in x is overwritten in Fig. 4.14(a). However, after SSA in Fig. 4.14(b), the high input $h1$ is leaking through $x1$. This can be tracked by our leak vector γ. Specifically, $\gamma_S = \langle\langle c\rangle, \langle x_h\rangle\rangle = \langle\langle\phi\rangle, \langle 0\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle x1_h, x2_h\rangle\rangle = \langle\langle\phi\rangle, \langle 1, 0\rangle\rangle$. Here, $k1 = k2 = 1$ and $h1 = 1$, then Eq. (4.6) expands into $(1-0) \not\leq 0$. Therefore, SSA is not relatively secure.

```
void foo(l1)
{
  x = h1;
  // use x;
  x = l1;
}
```

(a)

```
void foo(l1)
{
  x1 = h1;
  // use x1;
  x2 = l1;
}
```

(b)

Fig. 4.14: *An Example of Single Static Assignment*

4.9.1.3 Common Sub-expression Elimination

The compiler eliminates the common sub-expressions (CSE) to improve the performance by reducing the re-computation of the same expressions. In Fig. 4.15, the common sub-expression $a+b$ stored in t is used in the computation of both x and y. This transformation leads to a new leaky variable t when either x or y holds sensitive information.

Let the order of variables of S is $\langle a, b, c, d, x, y\rangle$ and of T is $\langle a, b, c, d, x, y, t\rangle$.] Specifically, $\gamma_S = \langle\langle c\rangle, \langle a_{h1}a_{h2}, \cdots, y_{h1}y_{h2}\rangle\rangle = \langle\langle\phi\rangle, \langle 10, 01, 00, 00, 11, 11\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle a_{h1}a_{h2}, \cdots, y_{h1}y_{h2}, t_{h1}t_{h2}\rangle\rangle = \langle\langle\phi\rangle, \langle 10, 01, 00, 00, 11, 11, 11\rangle\rangle$. Here, $|V_0| = 6, |V_1| = 7$ and $|I_h| = 2$, then Eq. (4.6) expands into $(1-1) \leq 0$ for both $h1$ and $h2$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1, |V_0| = 6, |V_1| = 7$, Eq. (4.7) expands into $4 - 3 \nleq 0$ and same holds for $j = h2$. Thus, it does not satisfy Eq. (4.7). Therefore, common sub-expression elimination is not relatively secure.

```
void foo()
{
  a = h1; b = h2;
  x = a+b+c;
  y = a+b+d;

}
```

(a)

```
void foo()
{
  a = h1; b = h2;
  t = a+b;
  x = t+c;
  y = t+d;
}
```

(b)

Fig. 4.15: *An Example of Common Sub-expression Elimination*

4.9.1.4 Loop-based Strength Reduction

Strength reduction in the loop replaces statements in the loop with less expensive operators. In Fig. 4.16, a multiplication operator is replaced by an addition operator. However, it introduces a new variable t in Fig. 4.16(b). When variable b is tainted or holds sensitive information in Fig. 4.16(a), strength reduction introduces a new leaky variable t in Fig. 4.16(b). Our leak vector γ can track these new leaky variables and show that strength reduction may be leaky. Specifically, $\gamma_S = \langle\langle c\rangle, \langle i, n, a, b\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle i, n, a, b, t\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1, 1\rangle\rangle$. Here, $|V_0| = 4, |V_1| = 5$ and $|I_h| = 1$, then Eq. (4.6) expands into $(1 - 1) \leq 0$ for $h1$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1$, Eq. (4.7) expands into $3 - 2 \nleq 0$. Thus, it does not satisfy Eq. (4.7). Therefore, strength reduction is not relatively secure.

```
b=h1;
for(i=1;i<n;i++)
{
    a[i]=a[i]+(b×i);
}
```

(a)

```
b=h1;
t=b;
for(i=1;i<n;i++)
{  a[i]=a[i]+t;
   t=t+b;  }
```

(b)

Fig. 4.16: *An Example of Loop-based Strength Reduction*

4.9.1.5 Loop Invariant Code Motion

When a computation inside a loop does not vary during loop iterations, the compiler may move these computations outside the loop. This is called loop invariant code motion. Fig. 4.17 shows an example in which a multiplication operation is moved outside the loop. However, this also introduces a new variable, t. When either x or y is tainted in Fig. 4.17(a), the new variable t is also tainted in Fig. 4.17(b) after code motion. Our leak vector γ can track these new leaky variables after the compiler applies loop invariant code motions.

Specifically, $\gamma_S = \langle\langle c\rangle, \langle i, n, a, x, y\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1, 0\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle i, n, a, x, y, t\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1, 0, 1\rangle\rangle$. Here, $|V_0| = 5, |V_1| = 6$ and $|I_h| = 1$, then Eq. (4.6) expands into $(1 - 1) \leq 0$ for $h1$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1$, Eq. (4.7) expands into $3 - 2 \nleq 0$. Thus, it does not satisfy Eq. (4.7). Therefore, code motion is not relatively secure.

```
x=h1;
for(i=1;i<n;i++)
{
   a[i]=a[i]+(x-y);
}
```

(a)

```
x=h1;
if(n>0)
   t=(x-y);
for(i=1;i<n;i++)
{ a[i]=a[i]+t; }
```

(b)

Fig. 4.17: *An Example of Loop Invariant Code Motion*

4.9.2 Secure Compiler Optimizations

We have analyzed a few other compiler transformations [21] like copy propagation, loop fusion, loop unswitching, loop unrolling, loop peeling, loop distribution, etc., and found that these transformations are relatively secure to the source program by our leak vector. A similar study can be performed for many other compiler optimizations as well.

4.9.2.1 Copy Propagation

The compiler removes redundant copies of variables by copy propagation. It reduces the register-to-register move instructions. In Fig. 4.18(a), variable y holds a copy of variable x. After the compiler applies copy propagation, y is removed in Fig. 4.18(b) and replaced by x at the use of y. As copy propagation removes the redundant copies of a variable, it does not lead to any extra information leak. Thus, it is relatively secure to the source program.

Our leak vector γ can track this relative security for high input h_1. Specifically, $\gamma_S = \langle\langle c\rangle, \langle x, y, z, a, b\rangle\rangle = \langle\langle \phi\rangle, \langle 1, 1, 1, 0, 0\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle x, z, a, b\rangle\rangle = \langle\langle \phi\rangle, \langle 1, 1, 0, 0\rangle\rangle$. Here, $|V_0| = 5, |V_1| = 4$ and $|I_h| = 1$, then Eq. (4.6) expands into $(1-1) \leq 0$ for $h1$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1$, Eq. (4.7) expands into $2 - 3 \leq 0$. Thus, it satisfies Eq. (4.7). Therefore, copy propagation is relatively secure.

4.9.2.2 Loop Fusing

Two loops having the same number of iterations with the same condition can be merged into one loop with all the loop statements. This is called loop fusion. However, if there is a statement $S2$ in the second loop dependent on a statement $S1$ in the first loop, it cannot be fused as this would lead to incorrect results in the fused loop. In Fig. 4.20(a), both the

```
void foo()
{
   x = h1;
   y = x;
   z = a+b+y;
}
```

(a)

```
void foo()
{
   x = h1;

   z = a+b+x;
}
```

(b)

Fig. 4.18: *An Example of Copy Propagation*

loops are fused into a single loop in Fig. 4.20(b). The loop statement in line 5 is dependent on line 2 in Fig. 4.20(a). Thus, these loops can be fused. The vice-versa situation is not possible for fusing, i.e., when the loop statement in line 2 is dependent on the statement in line 5. This loop transformation has the same fixed point by our leak vector before and after the transformation, as the loop statements are dependent in a continuous manner. Specifically, loop fusing can have only possible explicit leaks.

Here, $\gamma_S = \langle\langle c\rangle, \langle i, n, a, x\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle i, n, a, x\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1\rangle\rangle$. Here, $|V_0| = 4, |V_1| = 4$ and $|I_h| = 1$, then Eq. (4.6) expands into $(1 - 1) \leq 0$ for $h1$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1$, Eq. (4.7) expands into $2 - 2 \leq 0$. Thus, it satisfies Eq. (4.7). Therefore, loop fusing is relatively secure.

```
1 for(i=1; i<n; i++)
2 { a[i]=a[i]+h1; }
3
4 for(i=1; i<n; i++)
5 { x[i]=x[i]+a[i]; }
```

(a)

```
1 for(i=1; i<n; i++)
2 {
3      a[i]=a[i]+h1;
4      x[i]=x[i]+a[i];
5 }
```

(b)

Fig. 4.19: *An Example of Loop Fusing*

4.9.2.3 Loop Unswitching

When there is an invariant branch condition inside a loop, the compiler moves the branch outside the loop such that it replicates the loop body inside each branch of the condition.

It reduces the overhead of branching inside the loop. In Fig. 4.21(a), variable x is loop invariant. Thus, the loop is replicated to both the branches in Fig. 4.21(b). To ensure the execution of the loop inside the branch after the loop unswitching, the branch condition is guarded with another condition. This loop transformation has the same fixed point by our leak vector before and after the transformation as the data dependency, and the total iteration count does not change during the transformation.

Our leak vector γ can track these new leaky variables and show that strength reduction may be leaky. Specifically, $\gamma_S = \langle\langle c\rangle, \langle i, n, a, b, x\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1, 0\rangle\rangle$ and $\gamma_T = \langle\langle c\rangle, \langle i, n, a, b, x\rangle\rangle = \langle\langle \phi\rangle, \langle 0, 0, 1, 1, 0\rangle\rangle$. Here, $|V_0| = 5, |V_1| = 5$ and $|I_h| = 1$, then Eq. (4.6) expands into $(1-1) \leq 0$ for $h1$. So it satisfies Eq. (4.6). Now, we will check the sufficient condition for relative security. For $j = h1$, Eq. (4.7) expands into $2 - 2 \leq 0$. Thus, it satisfies Eq. (4.7). Therefore, loop unswitching is relatively secure.

```
for(i=1;i<n;i++)
{
   a[i]=a[i]+h1;

   if(x<10)
   b[i]=b[i]+a[i];
   else
   b[i]=b[i]-a[i];
}
```

(a)

```
if(n>0){
 if(x<10)
 {for(i=1;i<n;i++)
   {a[i]=a[i]+h1;
   b[i]=b[i]+a[i];  }}
 else
 {for(i=1;i<n;i++)
   {a[i]=a[i]+h1;
   b[i]=b[i]-a[i];  }}}
```

(b)

Fig. 4.20: *An Example of Loop Unswitching*

4.9.2.4 Loop Unrolling

Loop unrolling repeats the loop statements some number of times and iterates the loop for a reduced number of iterations. In Fig. 4.21, the loop is unrolled by two. After unrolling, the loop iteration is reduced to half as two iterations are executed at a time. It is relatively secure according to our leak vector, as data dependency of the loop statements is unaffected by loop unrolling.

```
for(i=2;i<n+1;i++)
{
   a[i]=a[i]+t;
}
```

(a)

```
for(i=2;i<n;i+2)
{
   a[i]=a[i]+t;
   a[i+1]=a[i+1]+t;
}
a[n]=a[n]+t;
```

(b)

Fig. 4.21: *An Example of Loop Unrolling*

4.9.2.5 Loop Peeling

A few number of iterations of the loop are peeled from the beginning or end of the loop. Loop peeling further improves loop fusion. In Fig. 4.22, the loop is peeled for one time from the beginning of the loop. Loop peeling is relatively secure according to our leak vector, as the data/control dependency does not change after the transformation. Though the iteration count is reduced due to peeling, it does not affect the total leak of the program. The fixed point of leak before the transformation would match with the total leaks due to the peeled statements from the loop and the fixed point of loop leak after the transformation.

```
for(i=2;i<n;i++)
{
      a[i]=a[i]+t;
}
```

(a)

```
a[2]=a[2]+t;
for(i=3; i<n; i++)
{
   a[i]=a[i]+t;
}
```

(b)

Fig. 4.22: *An Example of Loop Peeling*

4.9.2.6 Loop Distribution

The compiler applies loop distribution, which splits a single loop into multiple loops to improve the parallel execution of the loop statements. All the new loops have the same number of iterations. In Fig. 4.23, the two loop statements are distributed into two loops. Loop distribution is relatively secure according to our leak vector, as the total iteration count does not change during the transformation. The total leak due to the single loop before transformation would match with the total leaks due to each distributed loop after transformation. Thus, the fixed point of the original loop leak does not change.

```
for(i=2; i<n; i++)
{
    a[i]=a[i]+t;
    b[i]=b[i]+a[i];
}
```

(a)

```
for(i=2; i<n; i++)
{ a[i]=a[i]+t; }

for(i=2; i<n; i++)
{ b[i]=b[i]+a[i]; }
```

(b)

Fig. 4.23: *An Example of Loop Distribution*

4.10 Conclusion

Modern-day compilers apply various optimizations to improve the performance of the code in the target architecture. Compiler optimization can be functionally correct but does not always retain the security properties of the source program. In this thesis, we proposed a security measurement approach in a program with respect to information flow. Our approach finds both the explicit and implicit leaks in a program. Moreover, it finds the fixed point of the loop, which detects the maximum possible leakage inside the loop due to both explicit and implicit flows. We proposed three quantification parameters of information leakage in a program. We have defined the relative security between the two programs in terms of our quantification parameters. We have shown that the SPARK compiler is not relatively secure since it applies optimizations like DSE, SSA, CSE, code motion, etc.

5

TVIL: Translation Validation of Information Leakage of Compiler Optimizations

5.1 Introduction

Translation validation (TV) is the process of verifying that a source program is correctly translated into an optimized program by a compiler. A TV method can check the functional correctness and/or relative security of a translation process. In this thesis, we propose a translation validation method for the information leakage verification of compiler optimizations. A set of transformations is said to be secure if the amount of information leak in the optimized program is a subset of the amount of leak in the source program. Our translation validation method bi-simulates both source and optimized programs at the path level and propagates the information leaks to the subsequent paths recursively for checking the relative security between the programs. During translation validation, we measure the information leaks in paths using the technique proposed in our previous chapter. Our proposed method does not take any intermediate information from the compiler like the optimizations applied and the correlation of the variables in source and optimized programs. The threat model for this work is similar to our previous work in Chapter 4.

Threat model: We assume that the attacker can access the memory i) at the end of execution or ii) at some specific point of execution. The attacker can observe the values of all the program variables stored in the memory at the observation points mentioned above. Therefore, if a sensitive input gets leaked through a program variable, an attacker can always inspect the values of that variable to obtain the sensitive input. *To the best of our knowledge, this is the first work that proposes a translation validation method for checking the relative security of compiler optimizations with respect to information flow.* The unique contributions of this chapter are as follows:

- We formally define the relative security between source and optimized programs at path level based on the leak propagation vector.
- We then introduce a translation validation method based on a notion of leak propagation vector for checking the relative security between source and optimized programs.
- Correctness and complexity analysis of our method are also presented.
- The experimental results for various benchmarks in the SPARK compiler have been presented. The results show that the SPARK compiler does not retain the security properties of the source program.

The rest of the chapter is organized as follows. Section 5.2 presents the motivation behind our proposed work with an example. The translation validation approaches are presented in Section 5.3. Section 5.4 formulates the problem for the security of programs. The translation validation method for relative security is presented in Section 5.5. The soundness, termination, and complexity of our method are given in Section 5.6. The experimental results are shown in Section 5.7. Finally, Section 5.8 concludes the chapter.

5.2 Motivation

Functionally correct compiler optimizations may not always be secure. Let us consider the example of a source code and its optimized version after code motion in Fig. 5.1. In this example, the operation updating the variable b in blue has been moved to one of the branches, and the operations that update the variable z in red have been moved out of the branch. The code motion of ‘z’ reduces the code size, whereas the code motion of ‘b’

```
void f(l1,l2,h1,h2)
{
    x = h1;
    i = j = a = 0;
    while(i < l1)
    {
        b = a;
        a = a + h2;
        i++;
    }
    b = b - l2;
    if(i < x)
    {
        y = l1 + b;
        z = l1 + 5;
    }
    else
    {
        y = a + l2;
        z = l1 + 5;
    }
    if(j > z)
      c = l1 - 5;
    else
      c = l2 + 1;
    out = c + y + z;
}
```

(a)

```
void f'(l1,l2,h1,h2)
{
    x = h1;
    i = j = a = 0;
    while(i < l1)
    {
        b = a;
        a = a + h2;
        i++;
    }
    z = l1 + 5;
    if(i < x)
    {
        b = b - l2;
        y = l1 + b;
    }
    else
    {
        y = a + l2;

    }
    if(j > z)
      c = l1 - 5;
    else
      c = l2 + 1;
    out = c + y + z;
}
```

(b)

Fig. 5.1: *An Example of Conditional Speculation: (a) Source code, (b) Optimized code after code motion*

reduces the total number of instructions to be executed in run time on average. These code motions are correct since b is used only in one of the branches, and z is updating the same symbolic expression in both branches. Note that there may be further optimizations possible after this code motion. In these programs, $l1$ and $l2$ are low inputs, $h1$ and $h2$ are the high inputs, out is the output, and the rest are program variables. There is no leak through the variable z in the source code due to its same symbolic values in the parallel paths, and z does not leak in the optimized code as well. In the source code, the variable b leaks only $h2$ since a is leaking h_2 and b is explicitly dependent on a. In the optimized code, the variable b is leaking $h2$ in the same manner. In addition, b is also leaking $h1$ in the optimized code implicitly (since x of conditional expression is dependent on high input $h1$ and b has different symbolic values in the paths inside the conditional block), which is not there in the source code. Thus, b is leaking $h1$ in the optimized code, but there is no leak of $h1$ through b in the source code. Therefore, the optimized code is not relatively secure to the source code. The objective of this thesis is to validate the relative security between the source and optimized programs.

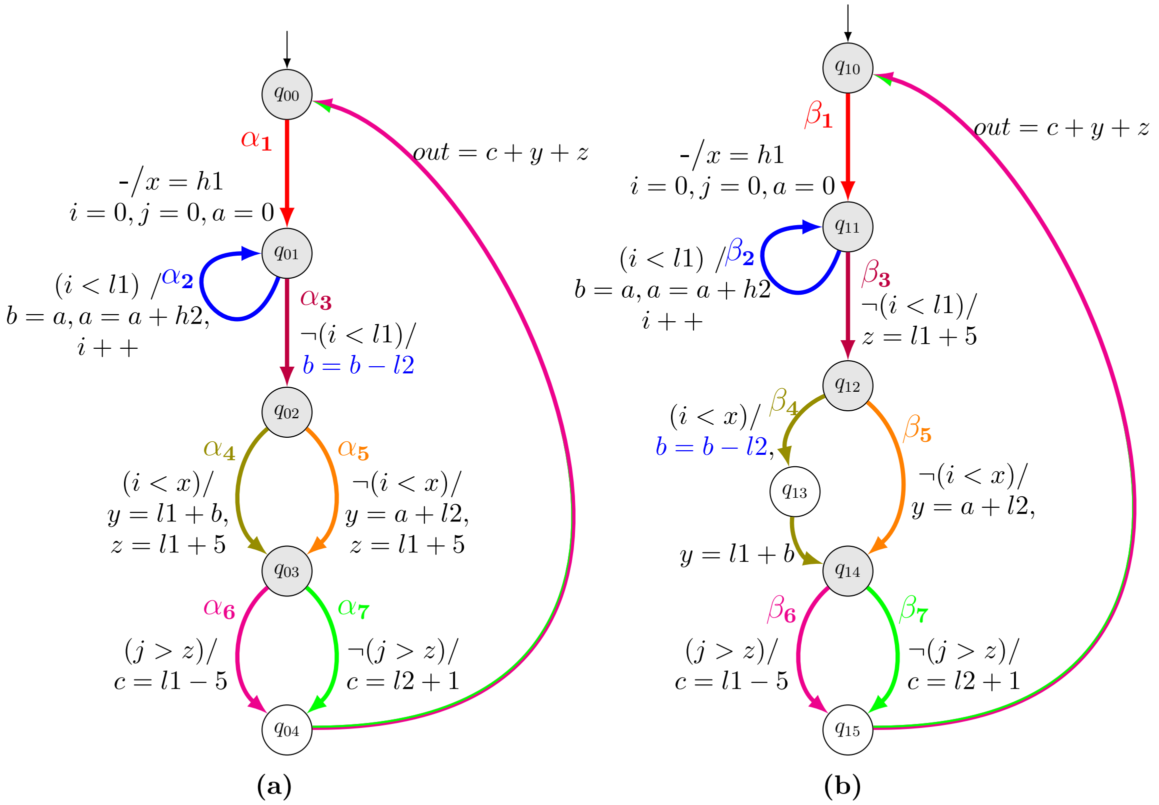


Fig. 5.2: *(a) Source FSMD M_0, (b) Corresponding optimized FSMD M_1*

5.3 Translation Validation Approaches

The relative security of two programs can be verified in three possible ways: (i) program level, (ii) trace level, and (iii) path level.

- In the program level TV method, we find the overall leak of both the source program and the optimized program independently and finally compare both the program leaks to verify the relative security between them. If the amount of leak in the optimized program is the subset of the leak in the source program, then the optimized program is relatively secure to the source program. The overall leak of both the source program and the optimized program can be measured using the method in Chapter 4. They can be compared to check the program level relative security. The issue with program level relative security is that there is no way to find the exact location of the leak, which might be responsible for the insecure compiler optimizations. Also, measuring the overall leak of a program is computationally intensive. The program level checking is particularly not effective in the context of compiler optimizations since the correlation of leaks between the source and optimized programs cannot be established in this method.

- In the trace level TV method, we will find all the traces in both source and optimized programs. We then find the leak of a trace in the optimized program and the leak of its corresponding trace in the source program. Then, we compare the leak to verify the relative security. This process is followed for all the traces in the optimized program. If for each trace in the optimized program, there exists a corresponding relatively secure trace in the source program, then the optimized program is relatively secure to the source program. However, the issue with the trace level relative security is the unbounded loops in the program, which may lead to a large or infinite number of traces in the program. For example, the while loop in Fig. 5.1 iterates for $l1$ times where $l1$ is an input. So, we need to find the trace for all possible values of $l1$. Thus, trace-level relative security is not useful in practice.

- In the path level TV method, cutpoints are inserted in a program, and the program is represented as a finite set of paths [1] between immediate cutpoints. In this process, the leak of a path in the optimized program and the leak of its corresponding path in the source program are obtained. The advantage of the path based approach is that the relative security between programs can be checked in a bi-simulation manner, and the

[1] As per the Definition 4.3.3, the cutpoints are the reset state, a loop entry point and branching states that post-dominates. Thus, the number of cutpoints is finite in a program. We consider paths from one cutpoint to another cutpoint without any intermediary occurrences of cutpoint. Hence, the number of paths is also finite.

exact location of the mismatch can be identified. For example, to verify the relative security between the FSMDs in Fig. 5.2, we start from the reset states and find the leak of corresponding paths β_1 and α_1. We compare the leaks of the corresponding paths and make a decision on either restarting the security validation process from their final states q_{11} and q_{01} or propagation of leaks to subsequent paths recursively to verify the relative security between the FSMDs. In many cases, the overall leak can be identified without traversing the entire programs as shown in this thesis.

In this thesis, our goal is to formalize the path level relative security and then develop a bi-simulation based translation validation method of relative security between the source and optimized programs.

5.3.1 Corresponding Paths

In this thesis, we represent a program as a finite state machine with datapaths (FSMD). The FSMD-based modelling is presented in Section 4.3 of Chapter 4. We are using the same definitions of path, trace, path cover, and cutpoints here. In addition, we find the corresponding paths and states in two FSMDs. Let the source and optimized FSMDs are represented as M_0 and M_1, respectively, and their path covers are represented as P_0 and P_1, respectively. We use a data-driven approach [122, 123] to find the correspondence of paths in P_0 and P_1. We find the potential corresponding paths (pc) α of M_0 and β of M_1 which is represented as $\alpha \simeq_{pc} \beta$ using data-driven approach in the following way:

1. Take a random input I and run on both source program M_0 and optimized program M_1.

2. Capture the traces τ_0 and τ_1, in M_0 and M_1, respectively for the input I.

3. Insert cutpoints based on Definition 4.3.3 in both τ_0 and τ_1.

4. Correlate the paths between cutpoints in τ_0 and τ_1, i.e., the corresponding paths.

5. Select the next input such a way that it traverses a new trace every time [121]. Repeat the above steps until the process covers all the paths in both M_0 and M_1.

The above-discussed data-driven approach works seamlessly if the control flow of the source program does not change due to the optimizations applied by the compiler, i.e.,

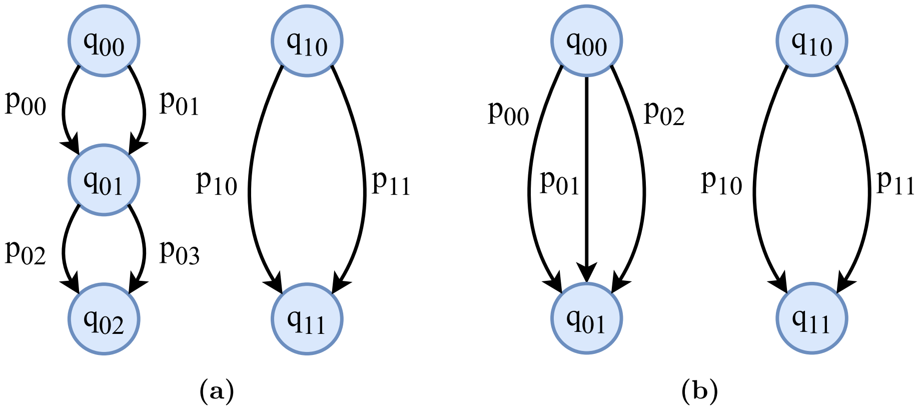


Fig. 5.3: *(a) Conditional block merging, (b) Parallel paths merging*

the total number of cutpoints remains the same during the optimizations. However, the following three scenarios may arise when the control flow changes due to loop merging, conditional block merging, or other optimizations.

- *Case1:* Two conditional blocks of the source program are merged into one. This scenario is presented in Fig. 5.3(a). In this case when p_{10} is traversed, initially p_{00} is chosen as a corresponding path. However, the execution data may not match for all the live variables at states q_{11} and q_{01}. Thus, it ignores the cutpoint q_{01} and considers the concatenated path $p_{00}p_{02}$ as the corresponding path. This is how it finds the other corresponding paths for conditional block merging.

- *Case2:* Two or more parallel paths of a conditional block are merged into one or vice-versa. This scenario is presented in Fig. 5.3(b). In this case, for one path there exists more than one corresponding path based on the maximum matching of execution data of live variables at that instance. For path p_{11} in Fig. 5.3(b), it may end up with two corresponding paths p_{01} and p_{02}.

- *Case3:* Two loops of the source program are merged into one in the case of loop fusions. This cannot be handled by our data-driven approach. Thus, our TV method is not applicable to such optimizations.

For the FSMDs in Fig. 5.2(a) and Fig. 5.2(b), the corresponding paths are $(\alpha 1, \beta 1)$, $(\alpha 2, \beta 2)$, $(\alpha 3, \beta 3)$, $(\alpha 4, \beta 4)$, $(\alpha 5, \beta 5)$, $(\alpha 6, \beta 6)$, and $(\alpha 7, \beta 7)$.

5.4 Security Problem Formulation

We have presented the program level relative security in Section 4.6.2 of the previous chapter with respect to the overall leak in the program. However, the limitations of the program level relative security, as discussed above, can be resolved as the path level relative security. Thus, we define path-level security first in this thesis. We then combine the concept of path cover and path level security to prove the relative security between the two programs. The theory behind our translation validation of the relative security of programs based on path level security is formulated below.

5.4.1 Security of Paths

The source FSMD M_0 and the optimized FSMD M_1 are $\langle Q_0, q_{00}, I, V_0, O, f_0, h_0\rangle$ and $\langle Q_1, q_{10}, I, V_1, O, f_1, h_1\rangle$, respectively. Here, V_0 and V_1 are the set of program variables in FSMD M_0 and FSMD M_1, respectively, I and O are the set of inputs and outputs for both M_0 and M_1, respectively. It may be noted that the input and output sets are the same for both M_0 and M_1 because M_1 is the optimized version obtained from M_0. However, we do not know the correlation between the variables in V_0 and V_1 since the optimization process is a black-box for us.

We consider two corresponding paths α and β in FSMDs M_0 and M_1, respectively. For two corresponding paths $\langle\alpha, \beta\rangle$; their respective start states, i.e., $\langle\alpha^s, \beta^s\rangle$ and final states, i.e., $\langle\alpha^f, \beta^f\rangle$ are corresponding state pairs. The leak vectors for paths α and β with initial leaks are represented as $\gamma_\alpha^{\alpha^s} = \langle\langle c_0\rangle, \langle n_{0v_0}, \cdots, n_{01}, n_{00}\rangle\rangle$ and $\gamma_\beta^{\beta^s} = \langle\langle c_1\rangle, \langle n_{1v_1}, \cdots, n_{11}, n_{10}\rangle\rangle$, respectively, where (i) $v_0 = |V_0 \cup O|$ and $v_1 = |V_1 \cup O|$, (ii) $c_0 = (b_{0_h} \cdots b_{0_1} b_{0_0})$, and $c_1 = (b_{1_h} \cdots b_{1_1} b_{1_0})$, (iii) $n_{0i} = (b_{0i_h} \cdots b_{0i_1} b_{0i_0})$, $1 \leq i \leq v_0$ and $n_{1i} = (b_{1i_h} \cdots b_{1i_1} b_{1i_0})$, $1 \leq i \leq v_1$. Here, (a) h is the number of high inputs, i.e., $h = |I_h|$, (b) b_{0i_j} and b_{1i_j} represents the leak value for i^{th} variable and j^{th} high input in $\gamma_\alpha^{\alpha^s}$ and $\gamma_\beta^{\beta^s}$ of paths α and β in FSMDs M_0 and M_1, respectively. The security of paths is defined as follows.

Definition 5.4.1 (Secure paths)**.** *The corresponding paths α of M_0 and β of M_1 are said to be secure if there is no leak in β and α. Formally, α and β are secure, denoted by $\beta \simeq_S \alpha$ (read as β is securely equivalent to α), if $\gamma_\beta^{\beta^s} = \gamma_\alpha^{\alpha^s} = \langle 0, \phi\rangle$.*

Definition 5.4.2 (Relatively secure paths)**.** *The corresponding paths α of M_0 and β of M_1 with leak vectors $\gamma_\alpha^{\alpha^s}$ and $\gamma_\beta^{\beta^s}$, respectively, are said to be relatively secure, denoted by $\beta \simeq_R \alpha$*

(read as β is relatively secure to α), if there is a leak in β, there must be a corresponding leak exists in α with the same number of leaky variables. Formally, α and β are relatively secure if they follow the necessary condition followed by the sufficient condition. The necessary condition is as follows:

$$\left(\bigvee_{i=1}^{V_1} b_{1i_j} - \bigvee_{i=1}^{V_0} b_{0i_j}\right) \leq 0,\ \forall j \in h \tag{5.1}$$

The above equation restricts any new leak of high input in β, which is not there in α. Also, it can verify whether β is more secure than α. The sufficient condition is as follows:

$$\left(\sum_{i=1}^{V_1} b_{1i_j} - \sum_{i=1}^{V_0} b_{0i_j}\right) \leq 0, \forall j \in h, s.t. \sum_{i=1}^{V_1} b_{1i_j} > 0 \tag{5.2}$$

The Eq. (5.1) satisfies when a high input leaking in β of M_1 is also leaking in α of M_0 irrespective of through how many low variables. The Eq. (5.2) satisfies when, for each high input, the number of variables leaking a high input in β is not more than that of in α. Thus, the sufficient condition needs to be checked only when the necessary condition holds.

Definition 5.4.3 (Conditionally secure paths)**.** *The corresponding paths α of M_0 and β of M_1, with leak vectors $\gamma_\alpha^{\alpha^s}$ and $\gamma_\beta^{\beta^s}$, respectively, are said to be conditionally secure, denoted by $\beta \simeq_C \alpha$ (read as β is conditionally secure to α), iff (i) $\beta \not\simeq_R \alpha$, (ii) $\alpha^f \neq q_{00}$ and $\beta^f \neq q_{10}$ and (iii) $\forall \beta'$ emanating from β^f, $\exists \alpha'$ emanating from α^f, such that $\beta' \simeq_S \alpha'$, $\beta' \simeq_R \alpha'$ or $\beta' \simeq_C \alpha'$,*

The Condition $\alpha^f \neq q_{00}$ and $\beta^f \neq q_{10}$ in definition 5.4.3 prevent leak propagation beyond reset state. It also ensures that if β' ends at reset state and there exists α', then $\beta' \simeq_S \alpha'$ or $\beta' \simeq_R \alpha'$ must hold. Specifically, paths terminating at reset states cannot be conditionally secure paths.

Now, based on the Definitions 5.4.1, 5.4.2 and 5.4.3, we formally define the corresponding state pair in FSMDs M_0 and M_1.

Definition 5.4.4 (Corresponding States)**.** *The corresponding state pair (CSP) between M_0 and M_1 is defined as follows:*

1. *The reset states q_{00} and q_{10} is a CSP, i.e., $\langle q_{00}, q_{10}\rangle$.*

2. *The final states $\langle \alpha^f, \beta^f \rangle$ of paths α with a leak vector γ^{α^s} at α^s and β with a leak vector γ^{β^s} at β^s is a CSP, if $\langle \alpha^s, \beta^s \rangle$ is a CSP and $\alpha \simeq_S \beta$ or $\alpha \simeq_R \beta$ or $\alpha \simeq_C \beta$.*

In an FSMD, a corresponding path pair $\langle \alpha, \beta \rangle$, can satisfy more than one security property upon different initial leak vectors. This is possible because the start states β^s and α^s may be reached with different predecessor paths and different propagated leak vectors. Thus, it may satisfy the following: i) $\beta \simeq_S \alpha$ and $\beta \simeq_R \alpha$ and $\beta \simeq_C \alpha$, if $\beta^f \neq q_{10}$ and $\alpha^f \neq q_{00}$, and ii) $\beta \simeq_S \alpha$ and $\beta \simeq_R \alpha$, if $\beta^f = q_{10}$ and $\alpha^f = q_{00}$. However, in a corresponding trace pair, the path pair $\langle \alpha, \beta \rangle$ always satisfies only one security property. This is due to the fact that a path in a trace is reached with a single predecessor path with a single propagated leak vector.

5.4.2 Relative Security of Programs

A program can be represented by a set of traces. The relative security of traces in two FSMDs is defined as follows:

Definition 5.4.5 (Relatively secure traces)**.** *Two traces $\tau_1 \in M_1$ and $\tau_0 \in M_0$ are said to be relatively secure, denoted as $\tau_1 \simeq_R \tau_0$ if it satisfies Eq. (5.1) followed by Eq. (5.2) for γ_{τ_1} and γ_{τ_0} (leak vectors for traces τ_1 and τ_0) considering the initial leaks at reset states q_{00} and q_{10} as Null.*

Based on the above definition, the following theorem captures the relative security between two behaviours.

Theorem 5.4.1. *An FSMD M_1 is said to be relatively secure to FSMD M_0, i.e., $M_1 \simeq_R M_0$, iff for each trace τ_1 in M_1, there exist a corresponding trace τ_0 in M_0 such that $\tau_1 \simeq_S \tau_0$ or $\tau_1 \simeq_R \tau_0$.*

However, due to unbounded loops, there may be a large or infinite number of traces in a program. Thus, finding all possible traces and checking their correspondence may not be feasible in practice. Therefore, by combining the Definition 4.3.2 and Theorem 5.4.1, the following theorem can be derived.

Theorem 5.4.2. *(Correctness of the approach): An FSMD M_1 with no unreachable states is said to be relatively secure to FSMD M_0, i.e., $M_1 \simeq_R M_0$ if for path cover $P_1 = [p_{1,1}, p_{1,2}, \cdots, p_{1,m}]$ of M_1, there exists a path cover $P_0 = [p_{0,1}, p_{0,2}, \cdots, p_{0,m}]$ of M_0, such that*

1. *for each path $p_{1,i} \in P_1$ with a propagated leak vector $\gamma^{p^s_{1,i}}$, $1 \leq i \leq m$, $\exists p_{0,j} \in P_0$ with a propagated leak vector $\gamma^{p^s_{0,j}}$, $1 \leq j \leq m$ s.t. $p_{1,i} \simeq_S p_{0,j}$ or $p_{1,i} \simeq_R p_{0,j}$ or $p_{1,i} \simeq_C p_{0,j}$ w. r. t. $\gamma^{p^s_{1,i}}$ and $\gamma^{p^s_{0,j}}$, ($\gamma^{q_{00}} = \gamma^{q_{10}} = \phi$). Also, the start states $\langle p^s_{0,j}, p^s_{1,i} \rangle$ and the final states $\langle p^f_{0,j}, p^f_{1,i} \rangle$ are in CSP,*

2. *if for a path $p_{1,i}$ with a propagated leak vector $\gamma^{p^s_{1,i}}$, $p^f_{1,i} = q_{10}$, then $\exists p_{0,j}$ with a propagated leak vector $\gamma^{p^s_{0,j}}$ s.t. $p^f_{0,j} = q_{00}$ and $p_{1,i} \simeq_S p_{0,j}$ or $p_{1,i} \simeq_R p_{0,j}$ w. r. t. $\gamma^{p^s_{1,i}}$ and $\gamma^{p^s_{0,j}}$,*

3. *if the paths $p_{1,k}$ and $p_{1,k+1}$ are consecutive in a trace, then $\gamma^{p^s_{1,k+1}} = \gamma^{p^f_{1,k}}, 1 \leq k < n$.*

Proof. Let us consider a trace τ_1 in M_1. Since P_1 is the path cover of M_1, τ_1 can be looked over as a concatenation of consecutive paths starting and ending at the reset state of M_1. Formally, $\tau_1 = [p_{1,i_1}, p_{1,i_2}, \cdots p_{1,i_n}]$, where $p_{1,i_k} \in P_1, 1 \leq k \leq n$, $p^s_{1,i_1} = p^f_{1,i_n} = q_{10}$ and $\gamma^{p^s_{1,i_{k+1}}} = \gamma^{p^f_{1,i_k}}, 1 \leq k < n$. From the hypothesis 1, there exists a sequence S_p of paths $[p_{0,j_1}, p_{0,j_2}, \cdots p_{0,j_n}]$, where $p_{0,j_k} \in P_0, 1 \leq k \leq n$ in M_0 such that $p_{1,i_k} \simeq_S p_{0,j_k}$ or $p_{1,i_k} \simeq_R p_{0,j_k}$ or $p_{1,i_k} \simeq_C p_{0,j_k}$, $1 \leq k < n$ and $p_{1,i_n} \simeq_S p_{0,j_n}$ or $p_{1,i_n} \simeq_R p_{0,j_n}$ w. r. t. propagated leak vectors $\gamma^{p^s_{1,i_k}}$ and $\gamma^{p^s_{0,j_k}}$. We need to show that, (i) S_p is also a trace τ_0 in M_0 and (ii) $\tau_0 \simeq_S \tau_1$ or $\tau_0 \simeq_R \tau_1$ w. r. t. propagated leak vectors as Null at respective reset states.

(i) We will show it by induction.

(Base case:) Assume the trace τ_1 is a single path p_1 in M_1 and there exists a corresponding path p_0 in M_0, s.t. $p_1 \simeq_S p_0$ or $p_1 \simeq_R p_0$ by hypothesis 2 because $p^f_1 = q_{10}$ and $p^f_0 = q_{00}$ w. r. t. $\gamma^{p^s_1} = \gamma^{p^s_0} = \phi$. This ensures $\tau_0 \simeq_S \tau_1$ or $\tau_0 \simeq_R \tau_1$.

(Inductive step:) Let us now assume that the initial l paths (say S_l and $l < n$) in sequence S_p are consecutive, i.e. the paths in S_p are consecutive. Thus, the final states $\langle p^f_{0,j_l}, p^f_{1,i_l} \rangle$ is a CSP (hypothesis 1). Let the updated leak vector at the final state of l paths is γ^0 in S_l. Now, we will consider the $(l+1)^{th}$ path (i.e. S_{l+1}) of S_p with initial leak vector γ^0. The paths $p_{1,i_{l+1}}$ of τ_1 with initial leak γ^1 (let) and $p_{0,j_{l+1}}$ (i.e., the final path in the sequence S_{l+1}) of S_p with initial leak γ^0 are the corresponding secure or relatively secure or conditionally secure paths and $\langle p^s_{0,j_{l+1}}, p^s_{1,i_{l+1}} \rangle$ is a CSP (hypothesis 1). Therefore $p^s_{0,j_{l+1}} = p^f_{0,j_l}$ since $p^s_{1,i_{l+1}} = p^f_{1,i_l}$ (τ_1 is a trace) and $\langle p^f_{0,j_l}, p^f_{1,i_l} \rangle$ is a CSP. Hence, the sequence S_{l+1} is also consecutive. By induction, the final state of last path in S_p i.e. $p^f_{0,jn} = q_{00}$ because the state pair $\langle p^f_{1,i_n}, p^f_{0,j_n} \rangle$ is a CSP and $p^f_{1,i_n} = q_{10}$. Therefore, by induction, S_p indeed is a trace τ_0 of M_0.

(ii) If the corresponding paths in τ_1 of M_1 and τ_0 of M_0 are either secure or relatively secure always, then M_1 is relatively secure to M_0 by hypothesis 1. Now, assume there is a corresponding conditionally secure paths in τ_0 and τ_1, i.e., $p_{1,i_k} \simeq_C p_{0,j_k}$, $1 \leq k < n$. Since τ_0 is a trace, the final paths ending at states p^f_{1,i_n} and p^f_{0,j_n} must be secure or relatively secure as it satisfies hypothesis 2 of the theorem, i.e., $p_{1,i_n} \simeq_S p_{0,j_n}$ or $p_{1,i_n} \simeq_R p_{0,j_n}$ which eventually ensures $p_{1,j_k} \simeq_C p_{0,i_k}$. Therefore, it ensures $\tau_1 \simeq_S \tau_0$ or $\tau_1 \simeq_R \tau_0$. □

It may be noted that the relative security of M_1 and M_0 discussed above are with respect to the path level leaks. However, the relative security discussion presented in Definition 4.6.2 is with respect to the program level leaks. The program level and path level translation validation approaches are discussed in Section 5.3.

```
Algorithm 8: $TranslationValidation(M_0,\ M_1)$
  Input: Source FSMD $M_0$ and optimized FSMD $M_1$
  Output: Whether $M_1$ is as secure as $M_0$ or not
  Insert cutpoints from set $CP_0$ in $M_0$ and $CP_1$ in $M_1$ and compute the set of path
    covers, $P_0$ and $P_1$ in $M_0$ and $M_1$, respectively.
  $CSP$ is the set of corresponding state pairs, $\Gamma$ is the set of calling state pairs, $Se$,
    $RS$, and $CS$ are the sets of secure path pairs, relatively secure path pairs, and
    conditionally secure path pairs.
  $CSP = \Gamma = \langle q_{00}, q_{10} \rangle$; $\gamma^{q_{00}} = \gamma^{q_{10}} = \langle 0, \phi \rangle$
  Initialize the sets Se, RS, and CS as null.
  $PreAnalysis(M_0, CP_0)$;
  $PreAnalysis(M_1, CP_1)$;
  foreach $\langle q_{0i},\ q_{1j} \rangle$ in $\Gamma$ where $q_{0i} \in Q_0$ and $q_{1j} \in Q_1$ do
      if $CheckUsage(\phi, q_{1j}, I_h)$ then
          if $ChkCorres(q_{0i}, q_{1j}, \gamma^{q_{0i}}, \gamma^{q_{1j}}, \Gamma, P_0, P_1, Se, RS, CS)$ fails then
              $M_1$ is not as secure as $M_0$;
              return failure;
          end
      end
  end
  return success;
```

5.5 Translation Validation Method for Relative Security of Programs

In this section, we develop the translation validation method for checking the relative security of two programs based on our Theorem 5.4.2. Specifically, we take the path covers of two FSMDs M_0 and M_1 and find the security relation between the corresponding paths of M_0 and M_1. The process starts from the resets state pairs and recursively bi-simulates both FSMDs to find the security between paths. The method relies on some look ahead properties to reduce the overall complexity of the proposed method. Finally, the method reports the relative security of M_1 with respect to M_0. The algorithm is discussed in detail below.

5.5.1 Algorithm Description

Our translation validation method is given as function *TranslationValidation()* in Algorithm 8. The function takes two inputs M_0 and M_1 which are the source FSMD and optimized FSMD, respectively. It first finds the path covers in an FSMD by inserting cutpoints. It stores the corresponding state pairs and calls state pairs in CSP and Γ, respectively. Also, it stores the corresponding secure, relatively secure, and conditionally secure path pairs in the sets Se, RS, and CS, respectively. It initializes both the CSP and Γ to reset state pairs $\langle q_{00}, q_{10} \rangle$ of M_0 and M_1, and updates the Γ dynamically to make the security verification efficient. Also, the initial leak vectors at reset states and the sets Se, RS, and CS are initialized to Null. The leak vectors are stored at each Γ. The translation validation method then calls the function *PreAnalysis()* for both FSMDs M_0 and M_1 to store the sets CV_0 and CV_1, the candidate variables for the implicit leak at each cutpoint of M_0 and M_1 respectively. For each state pair in Γ starting with the reset state pairs of M_0 and M_1, it calls the function *ChkCorres()* with the initial leaks at Γ to check if there exists a corresponding secure, relatively secure, or conditionally secure path, for each path emanating from these calling state pairs. The Algorithm 8 returns success when M_1 is relatively secure to M_0.

The function *ChkCorres()* takes the calling state pairs $\langle \alpha^s, \beta^s \rangle$, the initial leaks γ^{α^s} and γ^{β^s} at states α^s and β^s, respectively, the path covers P_0 and P_1 of both the FSMDs M_0 and M_1, respectively, the sets Se, RS, and CS as inputs and returns success when there exists a corresponding secure, relatively secure or conditionally secure path emanating from α^s in M_0 for each path emanating from the state β^s in M_1, otherwise returns failure. For

Algorithm 9: $ChkCorres(\alpha^s, \beta^s, \gamma^{\alpha^s}, \gamma^{\beta^s}, \Gamma, P_0, P_1, Se,\ RS, CS)$

```
Input: Two states $\alpha^s \in M_0$ and $\beta^s \in M_1$, the initial leaks $\gamma^{\alpha^s}$ and $\gamma^{\beta^s}$ at states $\alpha^s$ and $\beta^s$, two path covers $P_0$ of
       $M_0$ and $P_1$ of $M_1$, the sets $Se$, $RS$, and $CS$ for storing the secure, relatively secure, and conditionally secure
       paths.
Output: Returns success if for every path emanating from state $\beta^s$ there exists a secure, relatively secure, or
        conditionally secure corresponding path emanating from state $\alpha^s$, otherwise returns failure.
foreach path $\beta$ in $P_1$ emanating from $\beta^s$ do
    find $\alpha$ = corresponding path in $P_0$ emanating from $\alpha^s$;
    $\gamma_\beta^{\beta^s} = FindPathLeak(\beta, \gamma^{\beta^s})$;
    $\gamma_\alpha^{\alpha^s} = FindPathLeak(\alpha, \gamma^{\alpha^s})$;
    if $\beta \simeq_S \alpha$ // secure paths
    then
        $\gamma^{\alpha^f} = \gamma^{\beta^f} = \langle 0, \phi \rangle$;
        $Se = Se \cup (\alpha, \beta)$;  $CSP = CSP \cup (\alpha^f, \beta^f)$;
        $\Gamma = \Gamma \cup (\alpha^f, \beta^f)$;
    end
    else if $\beta \simeq_R \alpha$ // relatively secure paths
    then
        $RS = RS \cup (\alpha, \beta)$; $CSP = CSP \cup (\alpha^f, \beta^f)$;
        if $\alpha^f \neq q_{00}$ and $\beta^f \neq q_{10}$ then
            if $CheckUsage(\gamma_\beta^{\beta^s}, \beta^f, I_h)$ then
                $ChkCorres(\alpha^f, \beta^f, \gamma_\alpha^{\alpha^s}, \gamma_\beta^{\beta^s}, \Gamma, P_0, P_1, Se,\ RS, CS)$;
            else
                $\gamma^{\alpha^f} = \gamma^{\beta^f} = \langle 0, \phi \rangle$;
                $\Gamma = \Gamma \cup (\alpha^f, \beta^f)$;
            end
        end
    end
    else
        // checking for conditional secure
        $CCS = CCS \cup (\alpha, \beta)$; $CSP = CSP \cup (\alpha^f, \beta^f)$;
        if $\alpha^f \neq q_{00}$ and $\beta^f \neq q_{10}$ then
            if $CheckUsage(\gamma_\beta^{\beta^s}, \beta^f, I_h)$ then
                $ChkCorres(\alpha^f, \beta^f, \gamma_\alpha^{\alpha^s}, \gamma_\beta^{\beta^s}, \Gamma, P_0, P_1, Se,\ RS, CS)$;
            else
                $\gamma^{\alpha^f} = \gamma^{\beta^f} = \langle 0, \phi \rangle$;
                $\Gamma = \Gamma \cup (\alpha^f, \beta^f)$;
            end
        end
        else
            Report reset states reached and return failure;
        end
    end
end
$CS = CS \cup CCS$;
return success;
```

each such path β emanating from β^s, it finds the potential equivalent corresponding path α emanating from α^s. It calls the function *FindPathLeak()* to find the leak vectors of both β and α with initial leaks γ^{α^s} and γ^{β^s}, respectively.

The *ChkCorres()* checks for the three scenarios of security equivalence. If none of them

Algorithm 10: $FindPathLeak(\alpha, \gamma^{\alpha^s})$

```
Input: The path $\alpha$ and initial leak $\gamma^{\alpha^s}$
Output: Returns $\gamma_{\alpha}^{\alpha^s}$, the leak of path $\alpha$ with some initial leak at $\alpha^s$
// handle loop leak
if $\alpha^s = \alpha^f$                                  ▷ loop entry/exit state then
    if $(\alpha^s, \gamma^{\alpha^s})$ exists in LoopLeakList then
        $\gamma_{\alpha}^{\alpha^s} = \gamma_{\alpha}^{\alpha^s}$ for pair $(\alpha^s, \gamma^{\alpha^s})$ in LoopLeakList;
    else
        Store all the distinct parallel paths inside the loop emanating from $\alpha_0$ in List;
        $\gamma_{\alpha}^{\alpha^s} = FindLoopLeak(List, \gamma^{\alpha^s}, \langle 0, \phi \rangle)$;
        Store $\gamma_{\alpha}^{\alpha^s}$ for the pair $(\alpha^s, \gamma^{\alpha^s})$ in LoopLeakList;
    end
else
    // handle explicit leak
    $\gamma_{\alpha}^{\alpha^s} = FindExplicitLeak(\alpha, \gamma^{\alpha^s})$;
    // handle implicit leak
    if $\alpha^s$ is a branching state then
        $TC = CheckCulprit(\alpha, \gamma_{\alpha}^{\alpha^s})$;
    end
    if $TC \neq \phi$ and $CV_{\alpha^f} \neq \phi$ and $\alpha^f$ post-dominates $\alpha^s$ in $M$ then
        $\gamma_{\alpha}^{\alpha^s} = \gamma_{\alpha}^{\alpha^s} \vee FindImplicitLeak(\alpha, TC, CV_{\alpha^f})$;
    end
end
return $\gamma_{\alpha}^{\alpha^s}$;
```

satisfies, it returns failure, which implies M_1 is not as secure as M_0.

(i) When the corresponding paths β and α are secure (line 5), i.e., $\beta \simeq_S \alpha$, it updates the secure set Se with the paths β and α, and also updates the CSP and Γ with the final states β^f and α^f.

(ii) Otherwise, it checks for the relative security of β and α. If $\beta \simeq_R \alpha$ (line 11) and the final states are not the reset states of M_1 and M_0, it updates the relatively secure set RS with the paths β and α, and CSP with the final states β^f and α^f. When the paths are relatively secure, it calls the function *CheckUsage()* at the final state β^f to check for the use of any leaky variables (in $\gamma_{\beta}^{\beta^s}$) or use of any high input in any of the successor paths of β. The function *CheckUsage()* verifies the influence of the current leak in the successor paths of the FSMD. If *CheckUsage()* returns true, *ChkCorres()* calls itself recursively with the final states β^f and α^f, and the updated leak propagation vector $\gamma_\beta^{\beta^s}$ and $\gamma_\alpha^{\alpha^s}$ as its inputs. We describe the function *CheckUsage()* in detail later in Section 5.5.3. It may be noted that *CheckUsage()* is called only for M_1 because we are concerned about the relative security of M_1 with M_0. When $M_1 \simeq_R M_0$, vice versa may not hold, i.e., when M_1 is relatively secure to M_0, M_0 may not be relatively secure to M_1. Note that when $\beta \simeq_R \alpha$, we should not reset the leak as it may lead to false positive scenarios. Thus, we call *ChkCorres()* recursively for relatively secure paths to verify the influence of the current leak in the successor paths. However, we reset the leak vectors to Null when *CheckUsage()* returns false, i.e., there is no possible leak further in the successor paths. This prevents the successive recursive calls for the paths originating from the final states β^f and α^f with leak vectors $\gamma_\beta^{\beta f}$ and $\gamma_\alpha^{\alpha^f}$ in the FSMD M_1 and M_0, respectively. Moreover, this way, it reduces the time complexity of the overall validation method. It updates the Γ with the final states α^f and β^f so that the validation starts again from α^f and β^f.

(iii) If $\beta \not\simeq_R \alpha$ (line 23), it considers β and α as candidate for conditionally secure paths. and it updates the set CCS with the path pair $\langle \beta, \alpha \rangle$. The current paths are extended to check for the existence of any successor path that leads to a secure or relatively secure path. Then the function *CheckUsage()* is called if the final states β^f and α^f are not the reset states. Otherwise, it concludes the reset states reached with failure to find a corresponding secure path. If *CheckUsage()* returns true, then *ChkCorres()* calls itself recursively. If there is no further leak it resets the leak vectors to Null and updates the Γ. When the function *TranslationValidation()* returns successfully, it will ensure $\beta \simeq_C \alpha$. Thus, it updates the set CS with the candidate conditionally secure paths in CCS and returns success.

The function *ChkCorres()* calls the function *FindPathLeak()* for each corresponding path to find the leak of the path with some initial leak at the start state of the path. It takes a path α and the initial leak γ^{α^s} as inputs and returns the computed leak $\gamma_\alpha^{\alpha^s}$ as output. To minimize the re-computation for the fixed point of information leakage of loops, we keep track of the leak of loops in a *LoopLeakList* which stores $\langle \alpha^s, \gamma^{\alpha^s} \rangle$ pair at each loop entry point α^s given the initial leak γ^{α^s} which is already computed. Thus, at each loop entry state α^s, the function *FindPathLeak()* checks for the existence of a $\langle \alpha^s, \gamma^{\alpha^s} \rangle$ pair in *LoopLeakList* and updates the greatest fixed point of the loop, i.e., $\gamma_\alpha^{\alpha^s}$ from *LoopLeakList* if exists. Otherwise, the function *FindPathLeak()* finds the greatest fixed point of the loop for calculating the leak vector with respect to the new propagated leak at α^s. It returns the leak of the loop, which is stored in the leak vector $\gamma_\alpha^{\alpha^s}$. We assume that each loop is executed at least once and it always has a single exit path, i.e., there is no exit, break, return, or go to statements inside the loop. Thus, whenever a loop entry state of the FSMD is reached, the loop is traversed at least once before traversing the exit path of the loop. In the next recursive call of the function *FindPathLeak()* at the loop entry state α^s (with the propagated leak $\gamma_{\alpha}^{\alpha^s}$) it considers the exit path of the loop. Moreover, it considers the propagated leak of the loop as the initial leak for the exit path of the loop. When the path is not a loop, *FindExplicitLeak()* is called. After it calculates the explicit leak it calls the *CheckCulprit()* if the start state is a branching state and stores the set TC with all the influence high inputs in the condition of the branching state. When there exist candidate variables for the culprit branch, *FindImplicitLeak()* is called to measure the implicit leak of α. Finally, it returns the $\gamma_{\alpha}^{\alpha^s}$ i.e. the leak of the path α. The detail of the functions *FindExplicitLeak()*, *PreAnalysis()*, *CheckCulprit()*, *FindImplicitLeak()* and *FindLoopLeak()* can be found in our previous chapter. These functions are used in our translation validation approach to measure the leak in a path.

5.5.2 Attack Models

In this thesis, we consider two attack models, such that the attacker has access to the program variables: 1) at the end of the execution and 2) at each cutpoint. We propose two different translation validation methods for these two attack models. The overall translation validation method varies in finding the corresponding security of paths in the function *ChkCorres()*. The function *ChkCorres()* presented in Algorithm 9 and discussed in the previous subsection is basically for the first attack model. For the second attack model the function *ChkCorres()* executes as follows: we consider the attacker can observe the variables at the cutpoints in this case, thus, there is no point in checking conditionally secure paths when the paths are not relatively secure. At this point, the attacker can access the local memory to observe the newly introduced information leaks. Thus, for the second attack model, *ChkCorres()* does not execute the condition highlighted in yellow (line 23 of Algorithm 9) and returns failure if $\beta \not\simeq_R \alpha$ (line 11). Therefore, *ChkCorres()* returns success if for every path emanating from start state β^s there exists a corresponding secure or relatively secure path emanating from α^s.

It may be noted that the sets Se, RS, and CS are not mutually exclusive. A single path pair $\langle \beta, \alpha \rangle$ may be present in more than one set (i.e., Se, RS, and CS). This is due to the fact that the security of a path pair depends on their initial propagated leak, i.e., the start state of a path with some initial leak may be revisited with a different propagated leak (due to the excessive recursive calls of *ChkCorres()* function) which leads to different security of paths than before. For example, a path pair may be relatively secure with propagated leak $\langle \gamma_1^{\beta^s}, \gamma_1^{\alpha^s} \rangle$, but they become conditionally secure with propagated leak $\langle \gamma_2^{\beta^s}, \gamma_2^{\alpha^s} \rangle$. Thus, the path pair $\langle \beta, \alpha \rangle$ may be present in more than one set of Se, RS, and CS. A function call graph for the overall translation validation method is shown in Fig. 5.4. The correctness and complexity analysis for our translation validation method is presented in Section 5.6.

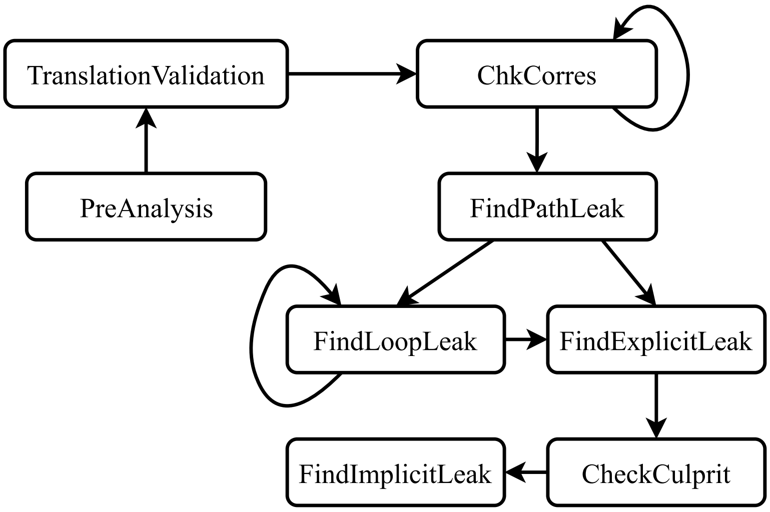


Fig. 5.4: *Function call graph for the translation validation method*

5.5.3 Minimizing Complexity by Look Ahead Properties

The function *ChkCorres()* calls itself recursively to verify the relative security of corresponding paths in both FSMDs. However, these recursive calls may lead to exponential complexity when the program contains a huge number of paths. Similar to Algorithm 7, we also use look-ahead properties here to minimize these recursive calls of the TV method. The function *CheckUsage()* at line 15 and line 26 in Algorithm 9 is used to minimize the total recursive calls of the overall translation validation method. The *CheckUsage()* function is called before each recursive call to the function *ChkCorres()*. It checks the impact of the current leak on the future paths of the optimized FSMD i.e., M_1. Specifically, it checks properties "v is used before definition" and “h is used”. This is achieved by applying the

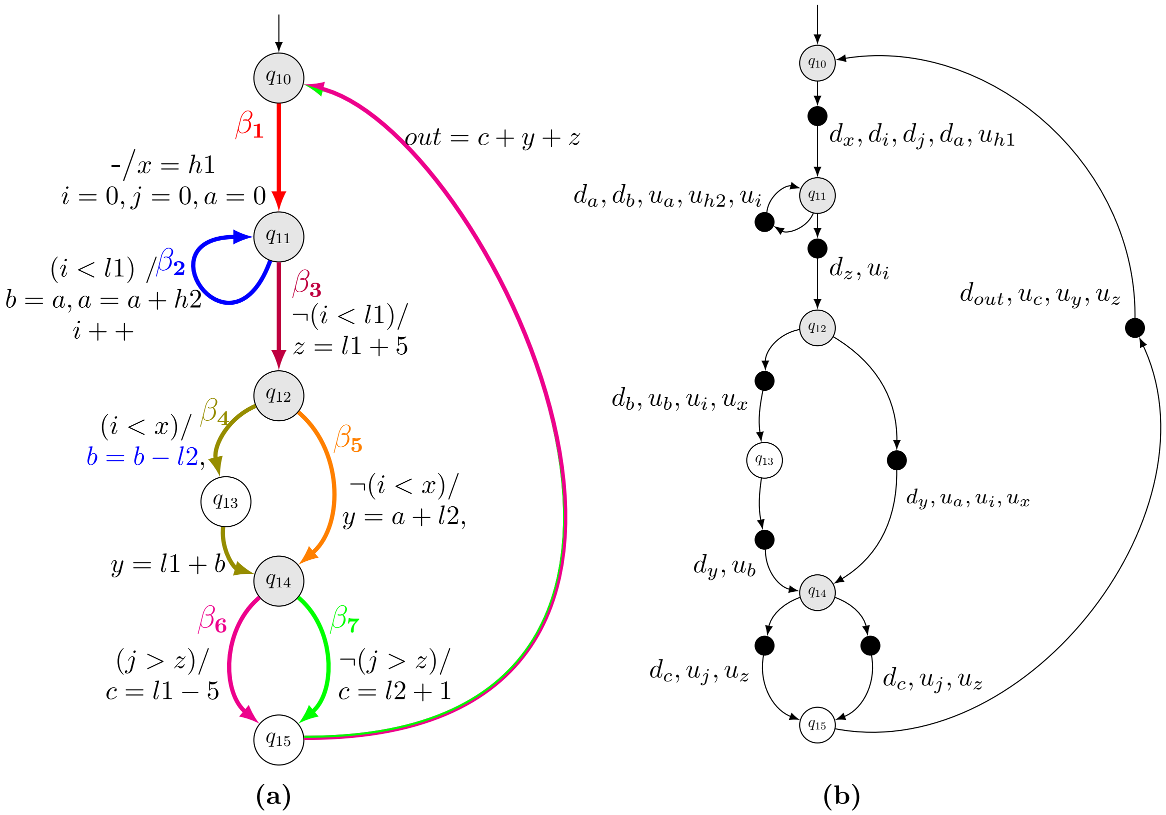


Fig. 5.5: *Kripke representation: (a) Optimized FSMD M_1 in Fig. 5.2(b), (b) Corresponding Kripke structure obtained from FSMD M_1*

CTL formulae $E[(\neg d_v)\ U\ u_v]$ and $EF(u_h)$ at the final state β^f for each leaky variable v in the leak vector $\gamma_\beta^{\beta^s}$ and for each high input $h \in I_h$, respectively. The function returns true if it finds the use of variable v before a definition of v in any successor path or the use of any high input h in any successor path. Note that we apply the CTL formulae only for FSMD M_1 as we are checking the relative security of M_1 with M_0 not vice-versa ($M_1 \simeq_R M_0 \not\Rightarrow M_0 \simeq_R M_1$). The *CheckUsage()* is explained in Section 4.5.2. The modeling an FSMD as the Kripke structure is also elaborated in Section 4.5.2 of that section. The Kripke structure for the FSMD in Fig. 5.2(b) is shown in Fig. 5.2(c). We have used the CTL model checker NuSMV [45] to verify the temporal property on the Kripke structure.

The leak vector γ at the CSP is Null when we call *CheckUsage()* at line 8 in Algorithm 8, i.e., there is no leak till this point during the bisimulation. Thus, here the function *CheckUsage()* only checks "h is used". This is achieved by applying the CTL formulae $EF(u_h)$ at the state $q_{1j} \in Q_1$ for each high input $h \in I_h$.

5.5.4 An Illustrative Example

Let us consider the example given in Fig. 5.2. Here, the work of our translation validation method is elaborated with this example.

The states q_{00} and q_{10} are the reset states in M_0 and M_1, respectively, and the states in gray are the cutpoints. The paths are shown with α_i and β_i, $1 \leq i \leq 7$ in M_0 and M_1, respectively. The *TranslationValidation()* starts with the $\langle q_{00}, q_{10}\rangle$ in Γ and calls the *CheckUsage()* at q_{10} to verify whether there is the use of any high inputs in future paths of the FSMD M_1. Here, *CheckUsage()* returns True as there is the use of $h1$ in β_1. Thus, it calls the *ChkCorres()*.

The *ChkCorres()* first takes the path β_1 and checks for the existence of a secure, relatively secure, or conditionally secure path in M_0. The leak vector of a path p in both M_0 and M_1 is represented as $\gamma_p = \langle\langle c_{h1}c_{h2}\rangle, \langle x_{h1}x_{h2}, y_{h_1}y_{h_2}, z_{h_1}z_{h_2}, a_{h_1}a_{h_2}, b_{h_1}b_{h_2}, c_{h_1}c_{h_2}, i_{h_1}\ i_{h_2}, j_{h_1}j_{h_2}, out_{h_1}out_{h_2}\rangle\rangle$. The explicit leak of β_1 is $\gamma_{\beta_1}^{\beta_1^s} = \langle\langle 00\rangle, \langle 10, 00, 00, 00, 00, 00, 00, 00, 00\rangle\rangle$. The corresponding path α_1 has also the same leak i.e. $\gamma_{\alpha_1}^{\alpha_1^s} = \gamma_{\beta_1}^{\beta_1^s}$. Thus, it must satisfy the conditions in Eq. 5.1 and Eq. 5.2. Hence, $\beta_1 \simeq_{RS} \alpha_1$.

Now, the *ChkCorres()* is called recursively for the path β_2 and its corresponding path α_2 which are loops. For both α_2 and β_2, there is a leak of $h2$ through variable a in the first iteration and through variable b in the second iteration. Thus, the loop leak for the paths β_2 and α_2 is $\gamma_{\beta_2}^{\beta_2^s} = \gamma_{\alpha_2}^{\alpha_2^s} = \langle\langle 00\rangle, \langle 10, 00, 00, 01, 01, 00, 00, 00, 00\rangle\rangle$. Thus, $\beta_2 \simeq_{RS} \alpha_2$. Similarly, the leak for the corresponding paths β_3 and α_3 is also the same as β_2 and α_2, respectively. Thus, $\beta_3 \simeq_{RS} \alpha_3$.

In the next recursive call, the implicit leak for β_4 is calculated since the branch is the culprit due to the presence of x in the condition of execution. The leak of β_4 is $\gamma_{\beta_4}^{\beta_4^s} = \langle\langle 10\rangle, \langle 10, 11, 00, 01, 11, 00, 10, 00, 00\rangle\rangle$. Note that y is explicitly dependent on b and is implicitly dependent on x (due to different symbolic values). Thus, there is an explicit leak of $h2$ through y and an implicit leak of $h1$ through y in β_4. Also, b has different symbolic values, so b is leaking $h1$ implicitly in β_4. The implicit leak for the corresponding path α_4 is $\gamma_{\alpha_4}^{\alpha_4^s} = \langle\langle 10\rangle, \langle 10, 11, 00, 01, 01, 00, 10, 00, 00\rangle\rangle$. This implies $\beta_4 \not\simeq_{RS} \alpha_4$, because Eq. 5.2 does not hold due to more leak in β_4 than α_4, i.e. b is leaking $h1$ in β_4 but not in α_4.

Thus, *ChkCorres()* will now check for the conditional security of β_4 and α_4 as the final states β_4^f and α_4^f are not the reset states. The *ChkCorres()* calls recursively for β_6 and its corresponding path α_6. The leak of path β_6 is $\gamma_{\beta_6}^{{\beta_6}^s} = \langle\langle 10\rangle, \langle 10, 11, 00, 01, 11, 00, 10, 00, 11\rangle\rangle$.

The leak of the corresponding path α_6 is $\gamma_{\alpha_6}^{{\alpha_6}^s} = \langle\langle 10\rangle, \langle 10, 11, 00, 01, 01, 00,\ 10, 00, 11\rangle\rangle$. Hence, $\beta_6 \not\simeq_{RS} \alpha_6$ as Eq. 5.2 does not hold (due to more leaky variables for high input $h1$ in M_1 i.e. b). Finally, *ChkCorres()* returns failure (line 35) as the final states β_6^f and α_6^f are the reset states. Thus, *TranslationValidation()* also returns failure as M_1 is not as secure as M_0.

5.6 Correctness and Complexity

In this section, we describe the soundness and termination of our method. The completeness of the translation validation method cannot be achieved since the problem is inherently undecidable [52,54,100]. We also analyze the complexity of our translation validation method.

5.6.1 Soundness and Termination

Theorem 5.6.1. *If Algorithm 8 returns successfully in line 15, then $M_1 \simeq_R M_0$.*

Proof. The sets Se, RS, and CS of secure, relatively secure, and conditionally secure pair of paths (from M_0 and M_1) are computed in lines 8, 13, and 24, respectively, of the function *ChkCorres()*. To ensure the correctness of the Algorithm 8, we need to show that (i) for each path $\beta \in P_1$ (of M_1), there exists a pair $\langle\alpha, \beta\rangle \in E = Se \cup RS \cup CS$, and (ii) if $\beta^f = q_{10}$, then $\langle\alpha, \beta\rangle \in Se \cup RS$. In other words, first, two hypotheses of Theorem 5.4.2 hold by E. We will prove this by contradiction.

Suppose a path β exists in P_1 that does not have a corresponding member in E. This indicates that the path β has not been considered during the execution of the Algorithm 8. This implies that the start state β^s is not considered during execution. However, there must be some other path β' that leads to the start state β^s as all the states are reachable. Let's consider the following cases:

$\beta' \in E$: There must be a pair $\langle\alpha', \beta'\rangle$ in E, where $\beta' \in P_1$ and either i) $\beta' \simeq_S \alpha'$ which implies the end state of β', i.e., β^s must belongs to the set Γ and β must have been considered eventually, as given in line 7 of *TranslationValidation()*, or ii) $\beta' \simeq_R \alpha'$ or $\beta' \simeq_C \alpha'$, which implies β would be definitely considered in some subsequent paths in the recursive call to *ChkCorres()* in line 16 or line 27 of *ChkCorres()*, respectively when the function *CheckUsage()* returns true. Otherwise, the end state of β', i.e., β^s must belong to the set

Γ and β must have been considered eventually, as given in line 7 of *TranslationValidation()* (contradiction).

$\beta' \notin E$: In this case, the function *ChkCorres()* is never called with the start state of path β'. If we continue this, we reach the paths emanating from reset state q_{10}. However, $\langle q_{00}, q_{10}\rangle \in \Gamma$ by line 3 of Algorithm 8. The lines 3 and 7 of Algorithm 8 and line 1 of *ChkCorres()* ensure that the reset state and the paths emanating from it must be considered which again leads to a contradiction.

Therefore, if Algorithm 8 terminates in line 9, hypothesis 2 of Theorem 5.4.2 must hold. To prove the second part, assume there exists a path $\beta \in P_1$ and $\alpha \in P_0$ such that $\beta^f = q_{10}$, $\alpha^f = q_{00}$ and $\beta \simeq_C \alpha$. In this case, the function *ChkCorres()* returns failure at line 35 (contradiction). □

Theorem 5.6.2. *The translation validation method for checking relative security always terminates.*

Proof. The functions *ChkCorres()* and *FindLoopLeak()* call themselves recursively with respect to the call graph in Fig. 5.4. Thus, to prove the termination of the translation validation method, we need to show these two recursive functions terminate. Let us assume that when the function *ChkCorres()* is called from line 9 in Algorithm 8, there are C cutpoints ahead of $\langle q_{01}, q_{1j}\rangle$. This implies it has $T = k + k^2 + k^3 + \cdots + k^C$ paths ahead, where k is the maximum number of paths between two consecutive cutpoints. When the *ChkCorres()* calls recursively (at line 16 or line 27), the cardinality of T decreases by 1. Note that the recursive calls do not go beyond reset states. In other scenarios, when the algorithm won't make further recursive calls (line 30 in Algorithm 9) having l cutpoints ahead, it effectively reduces $k + k^2 + k^3 + \cdots + k^l$ number of recursive calls. In this case, it reduces $k + k^2 + k^3 + \cdots + k^l$ paths from T. Since T is in the well-founded set of non-negative numbers having no infinite decreasing sequence, the Algorithm 9 cannot execute (any combination of) the recursive calls infinitely long.

The function *FindLoopLeak()* calls itself recursively when the leak vector of two consecutive iterations of the loop is mismatched (γ_{p_c} and γ_{loop}). Since the γ_{loop} is the union of all leaks identified so far, the number of 1's in it never decreases. Thus, in the worst case, the function calls recursively until each bit of γ_{loop_c} is 1. The γ_{loop_c} has $(\|V\|+1) \times \|I_h\|$ number of bits, where V is the set of variables and I_h is the set of high inputs. This implies the number of recursive calls is also finite. Thus, the function *FindLoopLeak()* always terminates.

Therefore, the translation validation method in Algorithm 8 always terminates. □

5.6.2 Complexity Analysis

The complexity of the overall translation validation depends on the following: 1) the complexity of checking the security of two corresponding paths of M_1 and M_0, and 2) the number of times the function *ChkCorres()* calls itself recursively. The first factor depends on the complexity of the function *FindPathLeak()* i.e., the time required to measure the explicit and implicit leak of path or a fixed point of leak of the loop in M_0 or M_1.

Assume there are C cutpoints, maximum k_1 parallel paths between two consecutive cutpoints, H high inputs, and V variables (including the outputs) in an FSMD. Assume there are k_2 parallel paths inside a loop. Then, for the first factor, i.e., to find the leak for the two given paths *FindPathLeak()* takes $O(C.V.k_1 + k_2^V.H.V)$ in the worst case (Refer Chapter 4 for details of the complexity analysis).

The function *ChkCorres()* calls itself recursively at each cutpoint for each path emanating from a cutpoint. So, it calls for $(k_1 + k_1^2 + \cdots + k_1^{C-1})$ times which is of the order of $O(k_1^C)$. Therefore, in the worst case, the complexity of the overall translation validation approach is $O(k_1^C(C.V.k_1 + k_2^V.H.V)) \simeq O(k^{(C+V)}.H.V)$ if we consider the $Max(k_1, k_2)$ as k.

When there is no recursive call for the function *ChkCorres()*, and there is a single path inside a loop, our translation validation method results in the best case. Thus, the best case scenario would be as follows: the translation validation method starts at the reset state pair, and all the corresponding paths β in M_1 and α in M_0 emanating from the reset state pair are secure, i.e., $\beta \simeq_S \alpha$. Also, the function *CheckUsage()* returns false at state β^f for all the parallel paths, i.e., there is no use of any high input in the successor paths. Thus, the product of time complexity for finding the leak of a path and the maximum number of parallel paths between the two consecutive corresponding state pairs starting from the reset state q_{10} results in the best case. Therefore, the best case complexity is $O(k.H.V)$.

5.7 Experimental Results

5.7.1 Setup

The proposed translation validation (TV) of checking the security of compiler optimization steps has been implemented in C. We have used the same test cases used in Section 4.8

obtained from the front end of SPARK [68]. We generate the FSMDs for both the source and optimized C code for each benchmark. In our tool, we automate the process of generating FSMD from a C code. The implementation flow of our proposed TV method is presented in Fig. 5.6. We have used an Intel Xeon(R) CPU E5-2620 v4 2.10GHz, 64GB of RAM, running Ubuntu 18.04.3 LTS in our experiments. Note that for the second attack model, i.e., when the attacker has access to the memory at cutpoints, the probability of information leak would be more than the first attack model due to more observation points for the attacker. Moreover, the run time would be faster in the case of the second attack model as it may identify the vulnerability at the earliest in most cases. hence, we present our results only for the first attack model, i.e., the attacker has access to the memory at the end of the execution.

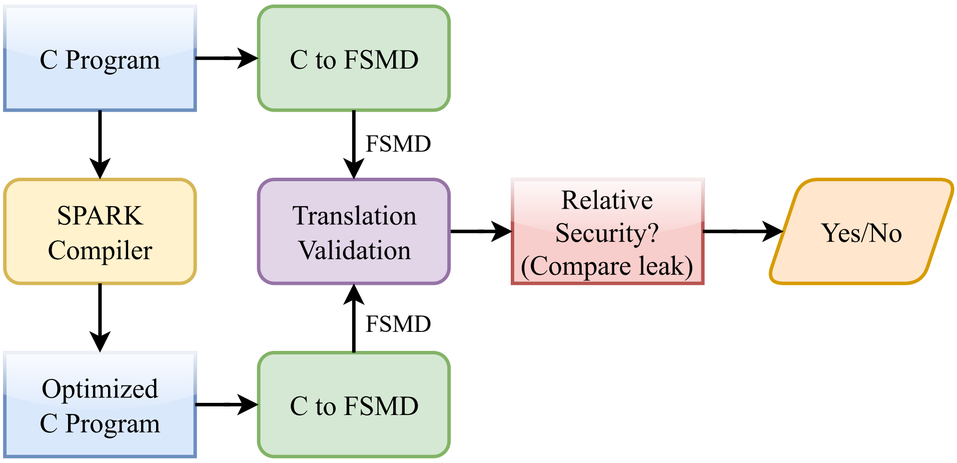


Fig. 5.6: *Overall flow for translation validation of information leakage*

5.7.2 Performance Measures

We present our experimental results for 12 benchmarks [22] in Table 5.1. We insert cutpoints into the FSMD to identify the paths. For each benchmark, the total number of cutpoints (#CP) and paths (#P) are presented in Columns 2 and 3, respectively. The SPARK tool does not change the control structure of the input code during its optimizations. Thus, for both source and optimized FSMD, the number of paths is the same. We represent the number of inputs for each benchmark in Column 4. We take a random subset of inputs as high inputs in our experiment, as presented in Column 5 (#H). We have considered different

Table 5.1: *Performance Measures for Benchmarks*

Bench	#CP	#P	#IN	#H	#RSP		RS?	ET (Sec)	
					NMC	MC		NMC	MC
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
dct	1	1	8	1	0	0	N	0.006	0.006
modn	2	10	1	1	0	0	N	0.01	0.01
diffeq	2	3	2	1	1	1	N	0.004	0.004
perfect	3	6	1	1	5	0	N	0.004	0.004
barcode	2	65	4	2	64	64	Y	0.076	0.073
parker	3	9	6	1	3	2	N	0.012	0.018
				1	5	3	N	0.012	0.016
				2	3	1	N	0.018	0.015
				2	1	1	N	0.013	0.009
find min8	8	15	8	1	12	4	Y	0.058	0.005
				2	12	2	N	0.114	0.021
				3	12	0	N	0.058	0.005
				5	8	0	N	0.018	0.001
waka	2	4	20	2	4	4	Y	0.007	0.005
				3	0	0	N	0.008	0.005
				4	4	4	Y	0.005	0.005
				5	1	1	N	0.005	0.005
lru	3	101	1	1	98	98	Y	0.066	0.078
qrs	13	56	1	1	56	50	Y	160.913	14.987
ieee754	6	519	2	1	518	518	Y	252.462	195.632
				1	516	516	Y	124.76	115.056

subsets of inputs as high inputs for some benchmarks like PARKER, FINDMIN8, WAKA, and IEEE754. The same number of high inputs actually represents a different subset of inputs. For example, PARKER has four rows in Column 5 as (1, 1, 2, 2), even though the first two rows have the same number as 1, but they actually represent different high inputs.

We have experimented the method of TV with and without applying the look ahead properties (discussed in Section 5.5.3). We report the total number of secure paths, relatively secure paths, and conditionally secure paths identified by the TV method. We found that the number of secure and conditionally secure paths is very small due to the leak of almost all chosen high inputs. Thus, we present only the relatively secure paths (#RSP) in Columns 6 and 7 with no model checking (NMC) and with model checking (MC) during the TV, respectively. Here, each path is counted exactly once to show the portion of an FMSD that is traversed to verify the relative security. It may be noticed that the total paths traversed by TV are reduced in 9 cases (out of 21 test scenarios) when we apply look ahead properties at the cutpoints just before the recursive calls. Note that when the TV returns failure due to a mismatch of leaks, the number in Columns 6 and 7 represents the paths traversed and found relatively secure till the point where the mismatch of the leak is found. These results show the effectiveness of our look ahead property checking during TV. The results for the relative security of the source and optimized benchmarks are presented in Column 8 (#RS). We found that in 13 out of the total 21 cases, the optimized program is not relatively secure to the source program. This highlights that modern compilers introduce security vulnerabilities in most cases. We have presented the average execution time (ET) in seconds for our TV with no model checking (NMC) and with model checking (MC) in Columns 9 and 10, respectively. We found that, except for QRS and IEEE754, other benchmarks take a negligible amount of time. The benchmarks QRS and IEEE754 take more time due to their large number of cutpoints and paths. However, for both benchmarks, the execution time is reduced significantly with MC. The average speed-up of the TV method with property checking is 3.67X times compared to the TV method without property checking.

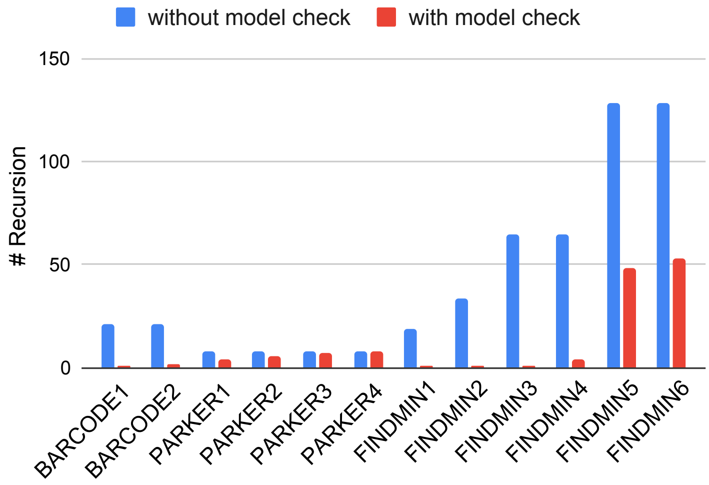


Fig. 5.7: *Number of Recursions with and without property checking*

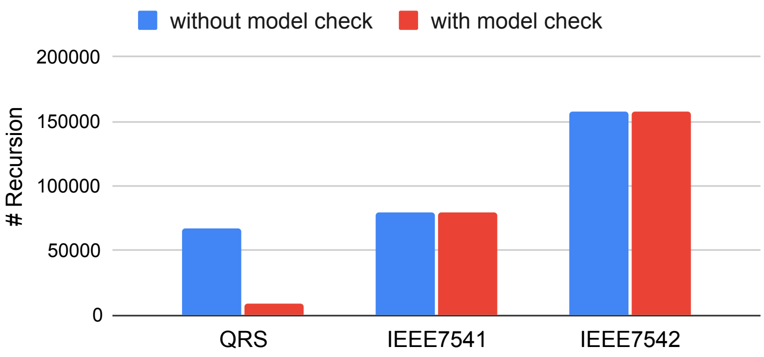


Fig. 5.8: *Number of Recursions with and without property checking for large benchmarks*

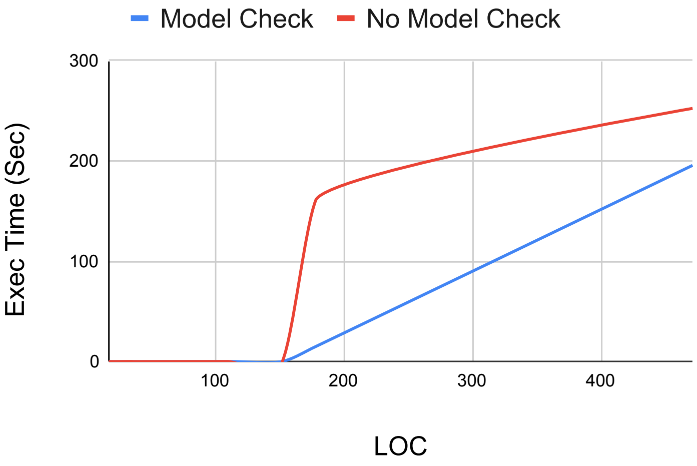


Fig. 5.9: *LOC Vs Execution Time (in Sec) with and without property checking*

5.7.3 Impact of Look-ahead Properties

In our implementation, we run each benchmark two times for a combination of high inputs (Column 5 in Table 5.1) with and without applying look-ahead properties, i.e., in Algorithm 8 for the first run, we call the function *CheckUsage()* in Line 8 of Algorithm 8, in Line 15, and Line 26 of Algorithm 9. Then, in the second run, we stop these function calls in our TV method. The objective is to check the impact of look-ahead properties on the total number of recursive calls in both runs. The total number of recursive calls of the function *CheckCorres()* without property check and with property check is shown in Fig. 5.7 and 5.8 for some significant benchmarks for which we found a difference. For better representation, we present the total recursions in two different figures due to the large variations in the number of recursions. For all the benchmarks in Fig. 5.7 and Fig. 5.8, it is observed that the number of recursive calls gets reduced significantly when look-ahead properties are applied. This shows the usefulness of the look-ahead property checking in our TV approach.

5.7.4 Scalability of Proposed Approach

We have presented the results for average execution time with respect to the lines of code (LOC) from our experiments in Fig. 5.9. It is apparent that the execution time is not growing exponentially with respect to the LOC in our experiments.

5.8 Conclusion

The compiler applies various optimizations to improve the performance in terms of area, power, etc. It is necessary to ensure the optimized program ensures the security properties of the source program in addition to the functional correctness. However, this is an undecidable problem. Thus, in this thesis, we used a concept of leak propagation with respect to information flow in a program to achieve the translation validation of information leakage of compiler optimizations. Moreover, we defined the relative security between the source and optimized program based on our leak propagation vector. We considered two different attack models and presented the translation validation method for them. Our experimental results concluded that the SPARK compiler is actually leaky and does not preserve the security of the source program in most cases. The proposed approach can be used by any compiler in its optimization phase to make it security aware such that it applies only the secure optimizations and restricts the optimizations leading to security vulnerability of the source.

6

MQIL: Model Checking based Quantification of Information Leakage in a Program

6.1 Introduction

The primary approach to identify information leaks in a program is taint analysis, which tracks how sensitive information flows through the program and if it is leaked to public observers. Taint analysis is not complete specifically for implicit information flow [37, 91]. Therefore, there may be false negative results. Also, it doesn't give a counter-example of the detected leaks in the program. On the other hand, bounded model checking is widely used for program verification and can provide counter-examples in case some property fails. Moreover, bounded model checking should be able to identify all information leaks for programs with static loop bounds. The motivational question is, can we model the quantification of information leakage in a program as a property verification problem in model checking? In this thesis, we measure the overall information leakage in a program with respect to information flow using the Bounded Model Checker for C programs (CBMC).

The threat model in this work is similar to that presented in Chapter 4. To the best of our knowledge, this is the first work that successfully quantifies the information leakage in any C program using CBMC and is scalable enough to handle large cryptographic benchmarks.

The major contributions of this chapter are as follows.

- We model the quantification method for information leakage in a program using CBMC.
- We show how to handle various constructs of a C program in our proposed method.
- We also show how to verify the relative security between a source and its optimized program using CBMC.
- We experiment with the proposed approach for various security benchmarks and show that it can successfully measure the overall information leakage in a program.
- We identify that LLVM actually introduces security vulnerabilities during optimization for most of the cryptographic benchmarks.

The rest of this chapter is organized as follows: Section 6.2 presents the motivation for the proposed approach. Our overall quantification approach is presented in Section 6.3. The proposed quantification model for various C constructs is presented in Section 6.4. The proposed quantification parameters and relative security are presented in Section 6.5. The experimental results are presented in Section 6.6. Finally, Section 6.7 concludes the chapter.

6.2 Motivation

Let us consider the function presented in Fig. 6.1. The function f() takes a low/non-sensitive inputs l, and a high/sensitive input h. Our focus is to measure the information flow from h to any of the program variables. The value of variable a depends on the value of high input h because, for two different values of h, the value of a also differs. Thus, there is an explicit information flow of h to a at Line 5 in Fig. 6.1(a). The variable d is dependent on both a and b. Since there is information flow to a from h, this implies there is an information flow to d as well from h. Similarly, the variable out has also an information flow from h through d. Thus, a, d, and out are the total leaky variables with respect to h for this given example. It may be noted that the value of b and c is not dependent on h. Thus, there is no information flow to b and c from h.

Our goal is to track these leaky variables using a model-checking method. Let us make two copies of the variable a as a_1 and a_2 and two copies of high input h as $h1$ and $h2$. The generated function with two copies of variables, high inputs, and assignment operations is presented in function f′() in Fig. 6.1(b). Note that the low input l has not been copied. Now, to consider different values of h let us assume $h1! = h2$ (Line 5), then the values of a are also not equal, i.e., $a1! = a2$. Since the values of a differ for two different values of h, the assertion at Line 16 will fail. Similarly, the assertions corresponding to d and out will also fail. For b and c, the assertions at Lines 17 and 18 will not fail. Thus, if we provide the f′() in Fig. 6.1(b) to CBMC, it will show that these three assertions failed. Thus, we can identify the information leakage using CBMC using the function f′() in Fig. 6.1(b). In this thesis, we develop a method to check explicit information flows, precise analysis of implicit information flows, and fixed points of leaks of the loops in a program as user-specified assertions to be verified by CBMC.

```
1  void f(l,h)
2  {
3        a = 10;
4        b = 20;
5        a = a + h;
6
7        b = b - l;
8
9        c = l - 5;
10
11       d = a + b;
12
13       out = c + d;
14
15
16
17
18
19
20
21 }
```

(a)

```
1  void f'(l,h1,h2)
2  {
3        a1 = 10; a2 = 10;
4        b1 = 20; b2 = 20;
5        assume(h1!=h2);
6        a1 = a1 + h1;
7        a2 = a2 + h2;
8        b1 = b1 - l;
9        b2 = b2 - l;
10       c1 = l - 5;
11       c2 = l - 5;
12       d1 = a1 + b1;
13       d2 = a2 + b2;
14       out1 = c1 + d1;
15       out2 = c2 + d2;
16       assert(a1==a2);
17       assert(b1==b2);
18       assert(c1==c2);
19       assert(d1==d2);
20       assert(out1==out2);
21 }
```

(b)

Fig. 6.1: *A Motivational Example: (a) Source code, (b) Generated source code*

6.3 Our Quantification Approach

Let $P(H, L, V, O)$ be a program, where H and L are the sets of high/sensitive and low/insensitive inputs, respectively, V and O are the sets of program variables and output variables, respectively. The program P consists of assignment statements, control blocks, loops, functions, and many more constructs. The overall quantification model to generate the program for CBMC is as follows.

- We create two copies of P at the abstract level as $P1(H1, L, V1, O1)$ and $P2(H2, L, V2, O2)$. Each element i in $H \cup V \cup O$ are copied as $i1$ and $i2$ in $P1$ and $P2$, respectively. It may be noted that we do not create copies of the low input set L.
- To quantify the information leakage, we add assertions of the form $v1 == v2, \forall v1 \in V1$ and $\forall v2 \in V2$, where $v1$ and $v2$ are corresponding to the variable $v \in V$ in the program P. Similarly, we add the assertions for the output variables as well.
- We assume that $h1! = h2 \, \forall h1 \in H1$ and $\forall h2 \in H2$, where $h1$ and $h2$ are corresponding to the high input $h \in H$ in the program P, i.e., each high input in P is assumed to take different values in $P1$ and $P2$.
- For different values of the high input h, if the assertion fails for a variable v, it implies that there is an information flow from h to v in the original program P.
- The combined code $P1$, $P2$, and all assumptions and assertions are given as input to CBMC. The total leak is defined by the number of asserts failed by CBMC.

The abstract idea of the proposed method for quantification of information leak is presented in Fig. 6.2. For the programs P_0 and P_1 in Fig. 6.3, the CBMC model is shown in Fig. 6.4.

6.4 Quantification Model for C Constructs

Although the core idea is to create two copies of the original program, there are various challenges to be addressed. For example, function calls cannot be copied and have to be handled in a different way to measure the leak propagation through function parameters and their return values. We now discuss how we handle the different C constructs in the program to create the input of CBMC.

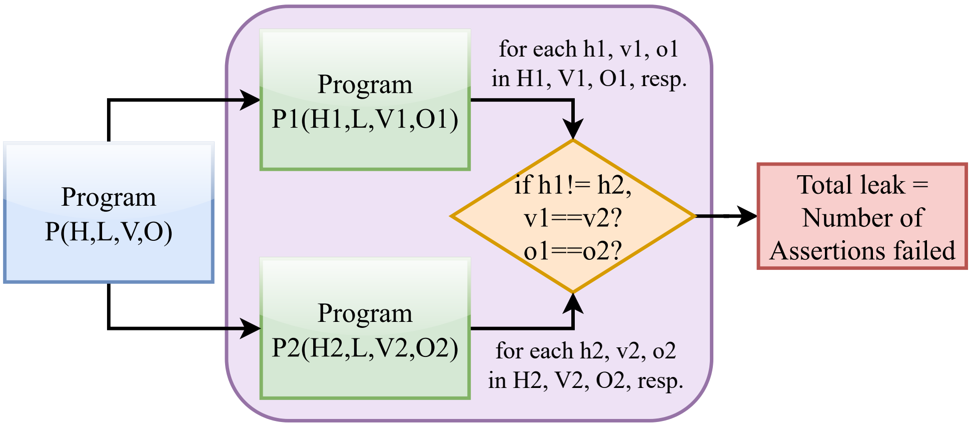


Fig. 6.2: *Overall flow of quantification model using CBMC*

6.4.1 Data types

6.4.1.1 Variables

Our aim is to verify the information flow from a sensitive or high input to any of the variables in the program. For each internal variable x, we create two copies, such as $x1$ and $x2$. The value of these two copies would be checked at the end of the program to verify the existence of information flow. If these values differ for a variable, this is identified as an information flow to the variable and marked as a leaky variable. For example, in Fig. 6.4, the copies $i1$ and $i2$ are created for variable i.

6.4.1.2 Structures and Unions

The members declared inside a structure or a union can be responsible for an information leak. Thus, we generate two copies of the structure variable or union variable instead of making copies of the members declared in a structure or union. For example, the object obj of structure in Fig. 6.5(a) is converted into objects $obj1$ and $obj2$ in Fig. 6.5(b).

6.4.1.3 Arrays

When one or more elements of the array have an information flow from any high input, it is considered a single leak of the array, as the rest of the elements of the array can also be accessible through the leaky element address. We generate two copies of the arrays, just like variables. For example, array $v[\,]$; in Fig. 6.4 is copied as $v1[\,]$, $v2[\,]$.

```
 1 void main(x,y,z,n,h)
 2 {
 3   // h is high input
 4   i = 1; u[0] = 0;
 5   b = x - y;
 6   c = x + y;
 7   while(i < n)
 8   {
 9     v[i] = u[i-1] + I;
10     u[i] = h;
11     i++;
12   }
13   if(u[i-1] > 0){
14     a = b - c;}
15   else{
16     a = b + z;}
17   out = v[i-1] + a;
18 }
```

(a) *Source program P_0*

```
 1 void main(x,y,z,n,h)
 2 {
 3   // h is high input
 4   i = 1; v[1] = 1;
 5   while(i+1 < n)
 6   {
 7     v[i] = h + i;
 8     i++;
 9   }
10   if(h > 0){
11     b = x - y;
12     c = x + y;
13     a = b - c;}
14   else{
15     b = x - y;
16     a = b + z;}
17   out = v[i-1] + a;
18 }
```

(b) *Optimized program P_1*

Fig. 6.3: *An Example of Conditional Speculation*

6.4.1.4 Pointers

A pointer variable may store the address of a leaky variable. We handle pointers the same way as variables to track the information flow through the address of any pointer, i.e., we generate two copies of the pointer variable of the same type.

6.4.2 Assignment Operations

An assignment operation is copied twice to track the information flow. The first assignment operation is with the first copy of variables and the second assignment operation is with the second copy of variables. For example, the assignment operation $a = b - c$ at Line 13 in Fig. 6.3(b) is copied twice at Line 15 and Line 22 in Fig. 6.4(b).

```
#include<assert.h>
void main(x,y,z,n,h1,h2)
{
  //h1 and h2 are high inputs
  __CPROVER_assume(h1!=h2);
  i1 = 1; i2 = 1;
  u1[0] = 0; u2[0] = 0;
  b1 = x - y;
  b2 = x - y;
  c1 = x + y;
  c2 = x + y;
  while(i1<n && i2<n)
  { v1[i1] = u1[i-1] + 1;
    v2[i2] = u2[i-1] + 1;
    u1[i1] = h1;
    u2[i2] = h2;
    i1++; i2++; }
  if(h1)
  { a1 = b1 - c1; }
  else
  { a1 = b1 + z; }
  if(h2)
  { a2 = b2 - c2; }
  else
  { a2 = b2 + z; }
  out1 = v1[i1-1] + a1;
  out2 = v2[i2-1] + a2;
  //quantify the leak
  assert(a1==a2); assert(b1==b2);
  assert(c1==c2); assert(i1==i2);
  assert(v1[i1-1]==v2[i2-1]);
  assert(out1==out2);
}
```

(a)

```
#include<assert.h>
void main(x,y,z,n,h1,h2)
{
  //h1 and h2 are high inputs
  __CPROVER_assume(h1!=h2);
  i1 = 1; i2 = 1;
  v1[1] = 1; v2[1] = 1;
  while(i1+1<n && i2+1<n)
  { v1[i1] = h1 + i1;
    v2[i2] = h2 + i2;
    i1++; i2++; }
  if(h1)
  { b1 = x - y;
    c1 = x + y;
    a1 = b1 - c1; }
  else
  { b1 = x - y;
    a1 = b1 + z; }
  if(h2)
  { b2 = x - y;
    c2 = x + y;
    a2 = b2 - c2; }
  else
  { b2 = x - y;
    a2 = b2 + z; }
  out1 = v1[i1-1] + a1;
  out2 = v2[i2-1] + a2;
  //quantify the leak
  assert(a1==a2); assert(b1==b2);
  assert(c1==c2); assert(i1==i2);
  assert(v1[i1-1]==v2[i2-1]);
  assert(out1==out2);
}
```

(b)

Fig. 6.4: *CBMC input for (a) the generated source program (GS) from P_0 (b) the generated optimized program (GOp) from P_1*

```
struct temp
{   int x;
    int a[10];
} obj;
```

(a) *Original Structure*

```
struct temp
{   int x;
    int a[10];
} obj1, obj2;
```

(b) *Generated Structure*

Fig. 6.5: *Handling Structure Construct*

6.4.3 Control Structures

The control structure in a program is responsible for implicit information flows. We generate two copies of the control block in a program. For example, for the control block presented in Fig. 6.3(b), the generated control block is presented in Line 12 to Line 25 in Fig. 6.4. When we assume the two copies of the high input h, i.e., $h1$ and $h2$ have different values, i.e., $h1! = h2$ (Line 5), it is possible that CBMC takes the true branch of the first copy of the control block and the false branch of the other control block or vice-versa during the execution of the program. Thus, in Fig. 6.4, the program execution generates different values for the copies of variables a and c, i.e., the assertions for $a1 == a2$ and $c1 == c2$ fails due to their different symbolic values. However, the assertion for variable b passes due to their same symbolic values in the parallel paths. Thus, only a and c are considered leaky variables due to the implicit flows. This is how our proposed approach analyzes the implicit flows in a program precisely and overcomes the problem of over-tainting as discussed in Section 6.2.

6.4.4 Loops

To find the fixed point of the leak of a loop, we generate two copies of the loop and merge them. Loop merging is done for the parallel execution of copies of the loop. However, the control blocks cannot be merged because the program execution needs to follow different parallel paths in the copies of the control block to track the implicit information flow in the program. This would not be possible if we merge the copies of the control blocks, as it will always take a single path. For the loop in Fig. 6.3(a), the generated loop is presented in Fig. 6.4 where the two copies of the original loop are merged.

Note that CBMC is a bounded model checker, thus, it cannot handle unbounded loops. Therefore, each loop in a program is bounded with a specific number of iterations to identify the fixed point of each loop. The option $--unwind\ k$ is used to fix the bound k for the loops in the program while running the CBMC. We perform a pre-analysis to find the minimum number of iterations a loop needs to iterate to find the fixed point of leak in the loop using the approach proposed in Chapter 4. A loop may have a number of parallel paths inside it due to control blocks. Let us consider a loop with two parallel paths, $P1$ and $P2$. In each iteration of the loop, it follows either $P1$ or $P2$. Then, our goal is to check all possible path sequences to get the fixed point of the leak in the loop, like $P1$, $P2$, $P1P1$, $P1P2$, $P2P1$, $P2P2$, $P1P1P1$, etc. It is possible that in each iteration, a new variable is introduced as a leaky variable. Thus, the depth or the maximum number of iterations for a loop would be the order of V in the worst case. We use this fixed point value as the unwind value k in CBMC for this loop. This will ensure that CBMC unrolls the loop enough number of times to identify all possible leaks inside the loop. The value of k is two for the loop in Fig. 6.3(b). The assertion for the copies of the array v in Fig. 6.4 fails due to the information flow from the high input h to the array v in the loop. It may be noted that most of the cryptographic benchmarks use bounded loops, thus, we do not need to find the fixed point of each loop individually to set the value of k, and CBMC verifies the assertions automatically by unrolling the loops.

6.4.5 Functions

For a function, information flow may occur through the local variables declared inside a function, function parameters, and/or function return value. To capture this, the function is handled in a different fashion. We generate two copies of each function definition f() in a program except the main function in the following way.

- *First copy (f()):* We use the original definition of the function as it is (with the original return statement, if any). This copy is used to track the information flow from the function parameters in the function to the value returned from the function. In the generated code, the first copy f() is called twice at the place of the original function call with the new copies of the argument variables. Consider the example of a program with a function call in Fig. 6.6(a) and Fig. 6.6(b). In the generated code, there are two function calls with the copies $h1$ and $h2$ as parameters in Fig. 6.6(c), at the place of the original function call f(h) in Fig. 6.6(a). Note that this copy does not quantify the information leakage within the function definition of f(). In the example, the leak

```
void main(h)
{
   int v;
   v=f(h);
}
```

(a)

```
int f(x)
{ int y;
  y=x+5;
  return y;
}
```

(b)

```
void main(h1, h2)
{assume(h1!=h2);
 int v1,v2;
 v1=f(h1);
 v2=f(h2);
 f'(h1,h2);
 assert(v1==v2);
}
```

(c)

```
void f'(x1, x2)
{
 int y1,y2;
 y1=x1+5;
 y2=x2+5;
 assert(y1==y2);
 assert(x1==x2);
}
```

(d)

Fig. 6.6: *Handling Function (a) Original main() function; (b) Original function definition f(); (c) Generated main() function; (d) Generated function definition f′()*

due to the variable v of the original $main()$ can now be tracked through the assert statement inserted in the generated $main()$ for the copies of the variable v as $v1$ and $v2$.

- *The second copy (f′()):* We copy the arguments and the local variables declared in the original definition of f(). This copy is used to identify the information leakage through the internal variables of the function and the function parameters, thus, it has no return value. The second copy f′() is called after the two function calls of f() in the generated code. For example in Fig. 6.6(c), function f′() is called after the two function calls of f(). It checks for the propagation of the leak to the local variables of the original function through the function parameters in the called function. Here, we add the assertions for each local variable and each function parameter at the end of the function definition f′(). For example, y is leaking in Fig. 6.6(a), so the assertion $y1 == y2$ will fail in this case.

6.5 Quantification Parameters and Relative Security

In Chapter 4, we proposed three quantification parameters for the overall information leak in a program:

- *Total leaky high inputs:* when there is more than one high input in a program, the leak of each high input is checked separately. Let us consider a program with two high inputs, $h1$ and $h2$. For the first parameter, to check the leak of $h1$, the two assumptions added are $h11! = h21$ and $h21 == h22$. Similarly, the leak of $h2$ is checked with assumptions $h11 == h21$ and $h21! = h22$. We assume the copies of one high input have different values and all other copies of high inputs have the same value. For each assumption, the assertions would be added for each variable. In Fig. 6.4(b) there is only high input h, the assertions fails for variables a, c, and v. Thus, h is considered as leaky and total leaky high input is one.

- *Unique leaky variables:* the total number of unique leaky variables measured is with respect to any high input. Here, a variable leaking more than one high input is considered a single leak. For the second parameter, to count the total leaky variables, the assumptions should be $h11! = h21$ and $h21! = h22$. We assume for each high input, the two copies of the high input are not equal. Then we add the assertions for each variable in the program. The total number of assertions added to the program is $|V|$ where V is the set of variables in the program. In Fig. 6.4(b), the unique leaky variables for P_1 is three.

- *Total leaky variables with respect to unique high input:* a variable leaking two high inputs is considered as two leaks. For the third parameter, to count the unique leaky variables with respect to $h1$, the assumptions are the same as for the first parameter. Then, we add the assertions for each variable of the program for each such assumption. Finally, we sum up the total failed assertions due to each individual high input in the program. Here, the total number of assertions added to the program is $|V| * |H|$, where V is the set of variables and H is the set of high inputs. In Fig. 6.4(b), the total leaky variables with respect to h is three.

All three parameters can be obtained in our CBMC based approach by just controlling the assumptions on the high inputs. With this setup, we verify whether any of the assertions failed. The high input $h1$ is said to be leaky if any of the added assertions fail for any variable in the program. The total number of failed assertions is the total number of leaky variables in the program.

6.5.1 Verifying Relative Security

Our proposed approach for quantification of leaks can be used to verify the relative security between a source program and its optimized version. We have taken the definition of relative security from Chapter 4 as follows.

Definition 6.5.1 (Relative Security)**.** *An optimized program is said to be relatively secure to the source program if it follows the following condition for each high input in the program. The number of variables leaking a specific high input in the optimized program is not more than the number of variables leaking the same high input in the source program.*

To verify the relative security, we find the third quantification parameter, i.e., the total leaky variables with respect to the unique high input for both the source and optimized programs. If for each high input, the total number of leaky variables is less than or equal to that of the source program, we say that the optimized program is relatively secure to the source program. For example, in Fig. 6.3, there is only one high input h, and it is leaked through a, u, and v in the source program P_0 and through a, c, and v in the optimized program P_1. Thus, the leak in P_1 is not more than the leak in P_0. Hence, this optimization is relatively secure.

6.6 Experimental Results

6.6.1 Setup

Our proposed approach is implemented in Python. The LLVM tool first generates an optimized LLVM IR from source C (S). Then, the tool llvm2c [11] is used to get an equivalent C (Op) from the IR. We have verified the functional correctness of both the source C and the optimized C using a simulation-based method. We use the C parser provided by the pycparser [9] to obtain the abstract syntax tree (AST) from S and Op. The proposed modification of C for quantification (ref. Fig. 6.4) is implemented at the AST level. The annotated C code, i.e., GS and GOp from S and Op, respectively, is generated by the C generator in the pycparser library from the modified AST. We used CBMC 5.10 [2] to quantify the information leak and verify the relative security between the source and optimized C code. The tool flow of our implementation is presented in Fig. 6.7. In our experiments, we used an Intel Xeon(R) CPU E5-2620 v4 2.10GHz, 64GB of RAM, running Ubuntu 18.04.3 LTS.

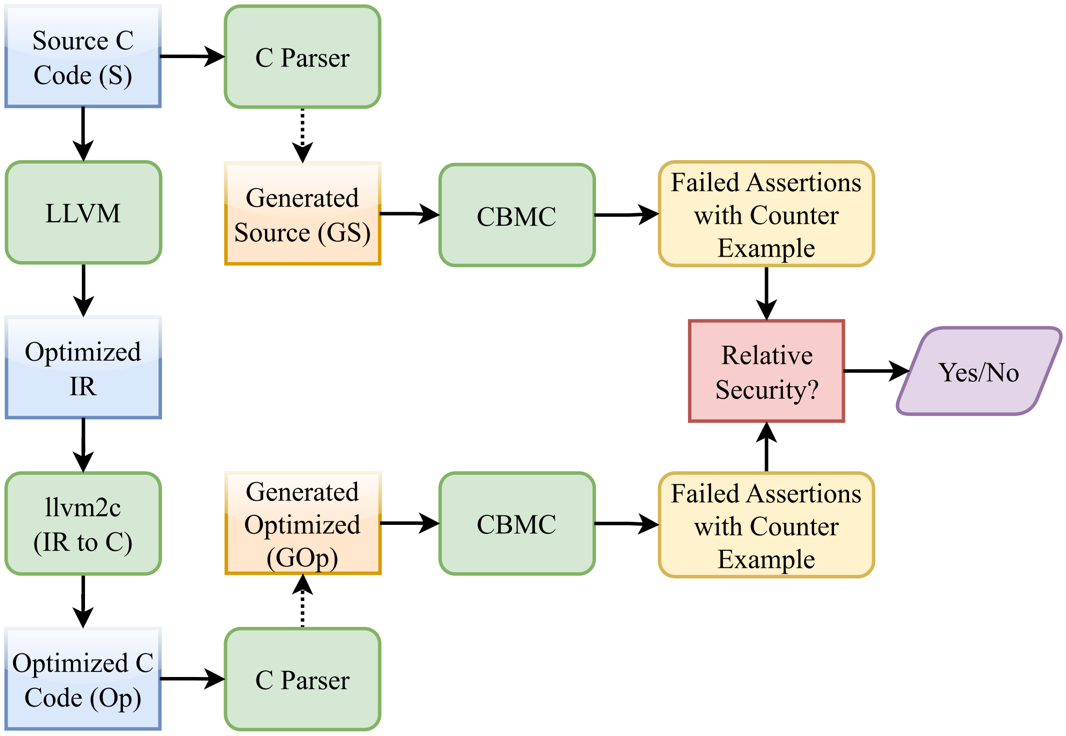


Fig. 6.7: *Implementation tool flow for quantification of information leak and relative security verification*

6.6.2 Benchmark Characteristics

We have presented the experimental results in Tables 6.1 to 6.3 for four cryptographic benchmarks, AES, DES, SHA, and RSA (taken from [1]). We considered the encryption mode for AES and DES, and the secret key is considered as high input for both benchmarks. The digest info is taken as high input for SHA to generate the hash value. RSA is an asymmetric cryptographic algorithm that uses both a public and private key for encryption and decryption, respectively. The private key is considered as the high input in our experiment. The characteristic of the source code (S) and the generated source code (GS) (i.e., the version of S that will input to CBMC) for each benchmark in terms of lines of code (#LoC), variables

Table 6.1: *Characteristics of Source code (S) and Generated Source (GS) code*

Bench	#LoC		#variable		#loop		#funccall		ET(Sec)
	S	GS	S	GS	S	GS	S	GS	
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
AES	358	1182	23	91	10	35	12	44	0.42
DES	247	782	44	150	20	58	7	26	0.17
SHA	171	565	30	97	12	31	13	42	0.25
RSA	66	136	13	45	3	6	4	15	0.04
Avg.	**2.95**×		**3.51**×		**2.74**×		**3.59**×		

(#variable), loops (#loop), and function calls (#funccall) is presented in Table 6.1. The #LOC, #variable, #loop, and #funccall are increased by 2.95, 3.51, 2.74, and 3.59 times, respectively, due to the two copies of the program. Although the loops are merged in the generated code, they are effectively copied if they are inside a control block. When a loop is inside a function, the number of loops is actually increased by three times as discussed in Section 6.4.5, and it may increase up to four times for nested function calls. Thus, the average number of loops in GS is more than two times that of S. The #funccall shown here excludes the assume and assertion functions in the GS. The values for the #funccall vary in the range of three to four times in GS than S. Our tool takes less than a second to generate GS from S, as shown in the last column of Table 6.1. Similar results are presented for the optimized code (Op) and the generated optimized code (GOp) in Table 6.2. The #LOC, #variable, #loop, and #funccall are increased by 3.83, 3.78, 1.49, and 3.44 times, respectively. The loops are unrolled in the optimization of RSA, thus, #loop in Op and GOp for RSA is 0. We observed that LLVM optimizes some of the function calls during the optimization, thus, #funccall is reduced in the Op than the S for AES, DES, and RSA, i.e., from 12 to 5, 7 to 4, and 4 to 1, respectively. Interestingly, we also observed that LLVM introduces new functions in its optimization phase, thus, the number of function calls is increased to 27 in the Op from 13 in S for SHA.

6.6.3 Performance Measures

We have presented the performance measures for the benchmarks in Table 6.3. For each benchmark, we have shown the number of inputs (#in) and high inputs (#high) in columns 2 and 3, respectively. The number of assumptions (#assume) inserted into the code is

Table 6.2: *Characteristics of Optimized (Op) and Generated Optimized (GOp) Code*

Bench	#LoC		#variable		#loop		#funccall		ET(Sec)
	Op	GOp	Op	GOp	Op	GOp	Op	GOp	
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
AES	998	3496	287	1620	10	22	5	19	1.12
DES	404	1425	149	467	14	41	4	14	0.46
SHA	479	1751	111	339	12	10	27	94	1.49
RSA	105	490	30	99	0	0	1	3	0.07
Avg.	**3.83**×		**3.78**×		**1.49**×		**3.44**×		

Table 6.3: *Performance Measures for Benchmarks in CBMC*

Bench	#in	#high	#assume	#assert		#leak		RS?	ET(Sec)	
				GS	GOp	GS	GOp		GS	GOp
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
AES	2	1	2	28	279	8	157	No	54.29	24.72
DES	3	2	5	57	150	14	66	No	21.55	14.55
SHA	2	1	2	32	142	3	11	No	0.31	3.09
RSA	5	1	5	5	21	2	1	Yes	0.01	5.58

presented in column 4. The inputs, high inputs, and assumptions are the same for both the generated source (GS) and the generated optimized code (GOp). The total number of assertions (#assert) inserted for each variable for each function in both the GS and the GOp is presented in columns 5 and 6, respectively. As LLVM IR is in SSA form, it introduces a large number of new variables during the optimization. Thus, the #assert is increased drastically for GOp than GS. The number of leaks in the respective program is the total number of assertions failed by the CBMC. The total number of leaks is presented in columns 7 and 8, respectively, for both GS and GOp. The relative security between the GOp and GS is presented in column 9. The results show that the optimized program is not relatively secure to the source program for all the cryptographic benchmarks in our experiment except RSA. This shows that LLVM introduces new security vulnerabilities in its optimization phase. The average execution time for running CBMC is presented in the last two columns for both the GS and the GOp, respectively. The results show that large benchmarks like AES and DES take less than a minute to quantify the leak. This signifies our proposed method is scalable enough to handle even large benchmarks. It may be noted that our proposed methods in Contributions 2 and 3 cannot handle any of the test cases presented here.

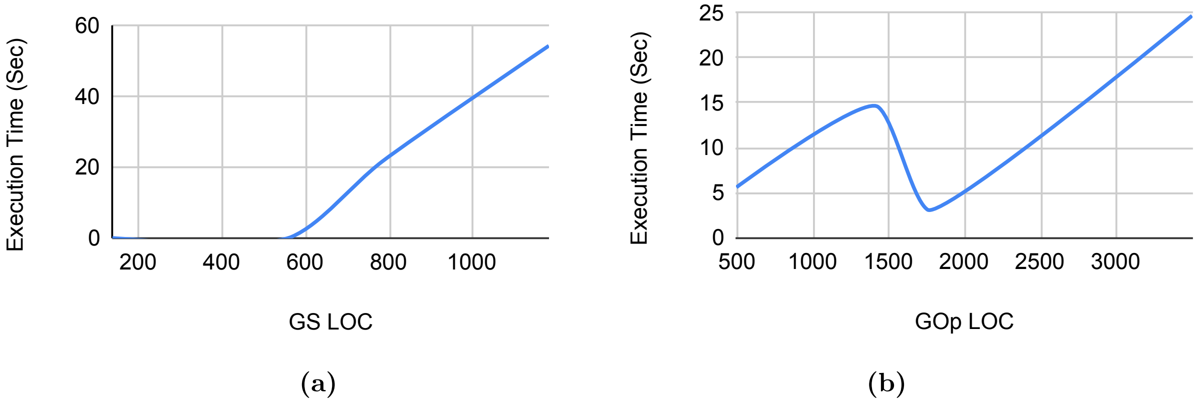


Fig. 6.8: *LoC Vs Execution Time (in Sec)*

6.6.4 Scalability of Proposed Approach

We have presented the results for average execution time with respect to the lines of code (LoC) from our experiments in Fig. 6.8(a) for the generated source (GS) and in Fig. 6.8(b) for the generated optimized code (GOp), respectively. It shows that the execution time is not growing exponentially with respect to the LoC in our experiments. Thus, the proposed approach is scalable enough to handle large benchmarks.

6.7 Conclusion

We proposed a quantification method for information leakage in a C program using the model checker CBMC. The proposed approach measures both the explicit leaks and the implicit leaks. Moreover, it handles the fixed point of the leak in a loop. The approach is further applicable to verify the relative security between a source program and its optimized version in a compiler. The experimental results show that the proposed approach is scalable. We observed that the LLVM compiler introduces new leaks in its optimization phase.

7

SRIL: Securing Registers from Information Leakage at Register Transfer Level

7.1 Introduction

Integrated Circuits (ICs) manufacturing is prone to defects or faults. Additional logic is usually added to the circuit to detect such faults through testing. Scan insertion is one of the techniques for design for testability (DFT) used to detect faults in the IC. During chip packaging, the scan chains are connected to external JTAG interface pins to provide on-chip debug capability and maintenance in the field. Access to the JTAG interface can be prevented by setting a protection bit. The scan chain can be left unbound to prevent further access. However, the protection bit can be compromised, and unbound scan chains can still be accessed by breaking the package open [133]. This scan access opens the gateway for the attackers to access the internal information to retrieve the secret information (keys) of the cryptographic designs.

Threat Model: We assume that the attacker has access to the scan chain of the functional IC (i.e., oracle) as mentioned above. Alternatively, he/she can also access the scan chain in the testing facility after the fabrication of the chip. But, he does not have access to the secret keys of the design. An attacker can observe the primary nonsensitive inputs, outputs, and the content of the intermediate registers through the scan chain inputs. His objective is to retrieve the secret keys from this intermediate register information available through the scan chain. In addition, this access to the functional IC and physical proximity to the design makes the attacker capable of performing power analysis attacks. An attacker can, through the statistical analysis of multiple power traces produced by multiple carefully planted plaintext (inputs to the algorithm), extract information that can reveal up to d intermediate values being processed by the circuit, where d is the attack order and is a measure of the capabilities of the attacker [72].

Due to the increasing complexity of the ICs, High-level Synthesis (HLS) is widely used for various applications (including cryptographic algorithms) to synthesize an RTL design from its behavioral specification. A shorter design cycle, the availability of behavioral specifications, fewer errors at higher abstraction levels, and the advancement of HLS technology enable users to move towards HLS for cryptographic designs [26, 94, 117]. Although there are efforts to track information leakage in the RTL generated by HLS [110], HLS does not protect against information leakage. However, protection against information leakage is desirable for cryptographic applications.

It has been shown that registers can leak information in a design [30, 59]. The objective of this thesis is to protect against information leakage through registers. In this thesis, we apply taint analysis at the source program to get the sensitive variables and obtain the mapped registers (sensitive) for these sensitive variables during the register allocation and binding phase of HLS. Finally, we introduce an RTL bubble-pushing algorithm to 'corrupt' the register content without impacting the final output of the design. To the best of our knowledge, this is the first work that proposes an HLS-driven register protection scheme against scan-based power side-channel attacks. Specifically, the contributions of this chapter are as follows:

- We propose bubble bubble-pushing approach for various components of the RTL design.
- We utilize the HLS flow to carry behavioral level precise taint analysis information into RTL.
- We then propose a method to thwart the information leakage of sensitive registers in RTL using the bubble-pushing algorithm.

- We experiment with our proposed approach on various benchmarks and show that it has a negligible performance overhead.
- An experiment on AES shows that bubble pushing actually protects against the power side channel.

The rest of the chapter is organized as follows. The HLS flow is presented briefly in Section 7.2. The proposed bubble-pushing approach for various RTL components is presented in Section 7.3. The overall flow of the proposed defense technique is presented in Section 7.4. Section 7.5 presents our experimental results, followed by the AES case study in Section 7.6. The discussion on some related queries is presented in Section 7.7. Finally, Section 7.8 concludes the chapter.

7.2 High-level Synthesis Flow

An HLS tool generates an RTL design from a source code using the following steps.

7.2.1 Preprocessing

The first step of HLS is preprocessing which takes advantage of the frontend of C/C++ compiler like GCC/LLVM. Using this compiler, it generates an internal representation (IR) from the input specification written in C/++. The front-end compiler also applies various optimizations on the IR. The HLS tool ignores the backend of such compilers, i.e., the machine instructions and register allocation steps. Instead, HLS performs the following steps on the IR obtained from GCC/LLVM. Consider the C code presented in Fig. 7.1(a). The corresponding IR in three address code is presented in Fig. 7.1(b).

```
g = a + a + b;
h = c + d + d;
```

(a)

```
e = a + b;
g = a + e;
f = c + d;
h = f + d;
```

(b)

Fig. 7.1: *An Example of a C Code and its 3-address code*

7.2.2 Scheduling

This step assigns the operations to the control steps, ensuring the data dependencies among operations. A control step corresponds to a clock cycle. As the source code is untimed, this step adds time to the design and identifies the execution clock cycle of each operation. This step aims to minimize the execution time of the design with limited resources. The corresponding scheduled data flow graph (DFG) for the IR in Fig. 7.1(b) is presented in Fig. 7.2. The first two addition operations $o1$ and $o2$ do not have any predecessors, thus, they are scheduled at time step s1 and the other two addition operations $o3$ and $o4$ are scheduled at time step s2 due to their predecessors $o1$ and $o2$ which are scheduled at time step s1. This schedule requires 2 time steps and 2 adders for successful design.

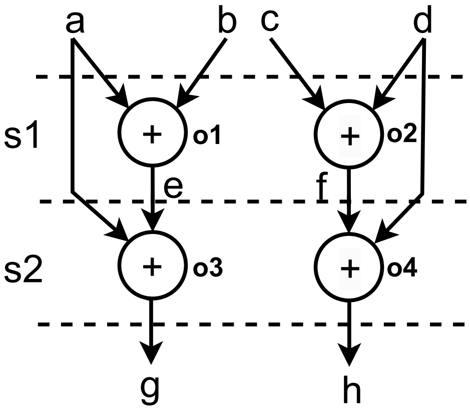


Fig. 7.2: *Scheduled DFG*

7.2.3 Allocation and Binding

The allocation process determines the types of required hardware components. Its objective is to determine the minimum resources required to satisfy the design constraint. After allocation, binding assigns the operations and variables to the allotted hardware components, i.e., functional units and storage elements (registers or memory). The functional unit binding for the scheduled behaviour in Fig. 7.2 is represented in Fig. 7.3. It requires two adders, four registers, and four multiplexers. The variables a and g are bound to register $r1$, the variables b and e are bound to register $r2$, the variables c and f are bound to register $r3$, and the variables d and h are bound to register $r4$. The addition operations $o1$ and $o4$ are bound to the adder circuit Add1 and $o2$ and $o3$ are bound to the adder circuit Add2.

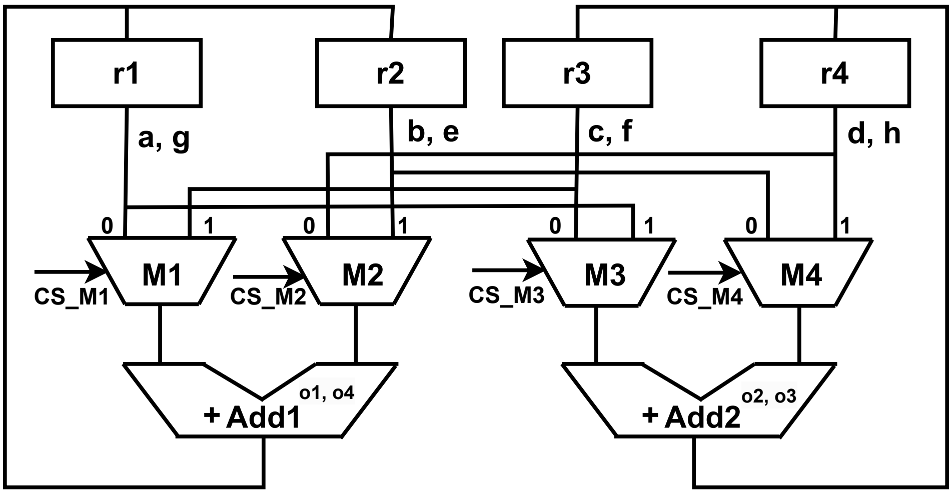


Fig. 7.3: *Functional unit binding with four multiplexers*

7.2.4 Datapath and Controller Generation

This step determines the interconnections among the functional units and registers. The datapath consists of a set of storage elements (like registers, memories, etc.), a set of functional units (like ALUs, multipliers, etc.), and interconnect elements (like multiplexers and buses). Each component may take one or more clock cycles to execute. A finite state machine (FSM) is generated for the control circuit which controls the execution of operations in the RTL design. To control the flow of data through the datapath the control unit generates control signals. The datapath and controller FSM for the scheduled design in Fig. 7.2 is shown in Fig. 7.3 and Fig. 7.4, respectively. The order of the control signal of multiplexers in Fig. 7.4 is $< CS\_M1\ CS\_M2\ CS\_M3\ CS\_M4 >$.

7.3 Proposed Bubble Pushing on RTL Circuit Components

The core idea of RTL bubble pushing is to corrupt the register content. For this purpose, we insert double inverters at the output of sensitive registers and push them randomly across the datapath components. This process is called "*bubble pushing*". These RTL datapath components will be modified with bubble pushing on the circuit. Consequently, the register will now store 'corrupt' data. Note that the functionality of the circuit does not change in the bubble-pushing process since we always insert an even number of inverters or bubbles.

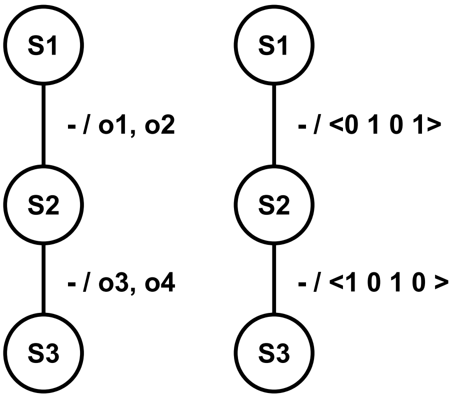


Fig. 7.4: *Controller FSM*

In this thesis, we study the impact of bubble pushing across RTL design components like logic gates, adders, subtractors, multipliers, multiplexers, and registers in the backward direction. The forward bubble-pushing approach can also be developed in a similar manner.

7.3.1 Logic gates

For the logic gates AND and OR, if there is a bubble at the output of these gates, the AND gate is changed to the OR gate, and the OR gate is changed to the AND gate after bubble pushing. The bubble pushing for the AND gate in Fig. 7.5(a) is shown in Fig. 7.5(b). For the XOR gate after bubble pushing, one of the inputs of the XOR gate gets a bubble. The bubble pushing for the XOR gate in Fig. 7.6(a) is shown in Fig. 7.6(b).

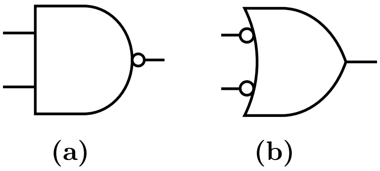


Fig. 7.5: *Bubble pushing on AND gate*

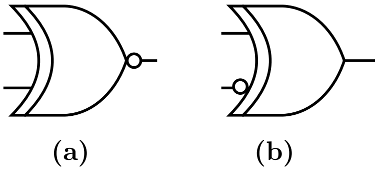


Fig. 7.6: *Bubble pushing on XOR gate*

7.3.2 Adder and Subtractor

Let us have two n-bit numbers a and b as inputs to an adder. The output of the adder stores into c, i.e. $c = a + b$. The negation of a number a can be represented in its 1's complement form, i.e. $\sim a = 2^n - 1 - a$. We find the updated design for the adder after bubble pushing as derived in Eq. 7.1.

$$\begin{aligned}
\sim a + \sim b &= 2^n - 1 - a + (2^n - 1 - b);\\
&= 2^{n+1} - 1 - a - b - 1;\\
&= 2^n - 1 - a - b - 1;\\
&= \sim (a+b) - 1;\\
\sim (a+b) &= \sim a + \sim b + 1;
\end{aligned} \tag{7.1}$$

Note that the overflow of a n-bit number is represented as a n-bit number, i.e. 2^{n+1} is represented as 2^n. Let the output of a subtractor be $c = a - b$. We find the updated design for the subtractor after bubble pushing as derived in Eq. 7.2. The bubble-pushing approach for an adder and subtractor is illustrated in Fig. 7.7. The designs in Fig. 7.7(b) and Fig. 7.7(d) are the updated designs after bubble pushing for the adder and subtractor in Fig. 7.7(a) and Fig. 7.7(c), respectively.

$$\begin{aligned}
&\sim a - \sim b = 2^n - 1 - a - (2^n - 1 - b);\\
&\sim a - \sim b = (2^n - 1 - (a - b)) - 2^n + 1;\\
&\sim a - \sim b = \sim (a - b) - 2^n + 1;\\
&\sim (a - b) = 2^n - 1 - (\sim b - \sim a);\\
&\sim (a - b) = \sim (\sim b - \sim a);
\end{aligned} \tag{7.2}$$

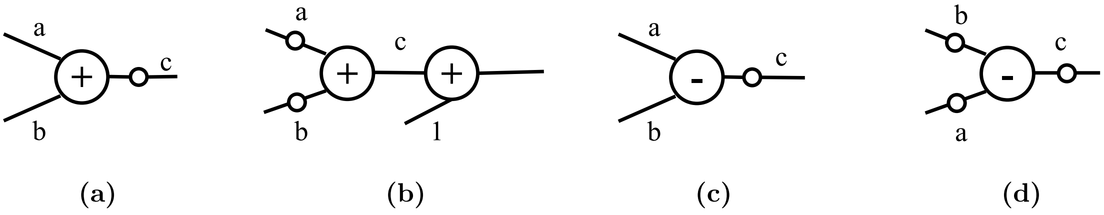
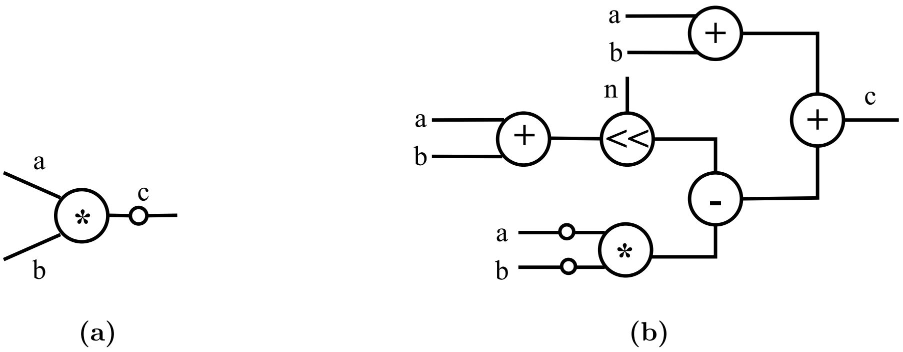


Fig. 7.7: *Bubble pushing on Adder and Subtractor*

7.3.3 Multiplier

Let us have two n-bit numbers a and b as inputs to a multiplier. The output of the multiplier stores into c, i.e. $c = a * b$. We find the updated multiplier circuit after bubble pushing as derived in Eq. 7.3. In the RTL code, the multiplication with 2^n is performed by left shifting the operand to n bits. The updated design for the multiplier in Fig. 7.8(a) is shown in Fig. 7.8(b).

$$\begin{aligned}
\sim a* \sim b =& (2^n - 1 - a) * (2^n - 1 - b);\\
\sim a* \sim b =& 2^n * (2^n - 1 - b) - (2^n - 1 - b)\\
& - (2^n * a) + a + (a * b);\\
\sim a* \sim b =& 2^n(2^n - 1 - (a + b)) - (2^n - 1 - (a * b))\\
& + (a + b);\\
\sim a* \sim b =& 2^n(\sim (a + b)) - (\sim (a * b)) + (a + b);\\
\sim (a * b) =& 2^n(\sim (a + b)) - (\sim a* \sim b) + (a + b);
\end{aligned} \tag{7.3}$$



(a) **(b)**

Fig. 7.8: *Bubble pushing on Multiplier*

7.3.4 Multiplexer

A multiplexer (Mux) with n select lines and 2^n inputs is shown in Fig. 7.9. When there is a bubble at the output of Mux as in Fig. 7.9(a), the bubble is pushed to all the inputs of the Mux from the output as shown in Fig. 7.9(b). In RTL code a 2:1 Mux with reg2 and reg3 as inputs and reg1 as output is represented using the ternary operator as *reg1 = (condition)?reg2:reg3*. If there exists a bubble at the output of Mux i.e. *reg1 =* *∼((condition)?reg2:reg3)*. After the bubble is pushed to the multiplexer in the backward direction, the updated code is represented as *reg1 = (condition)? ∼ reg2: ∼ reg3*.

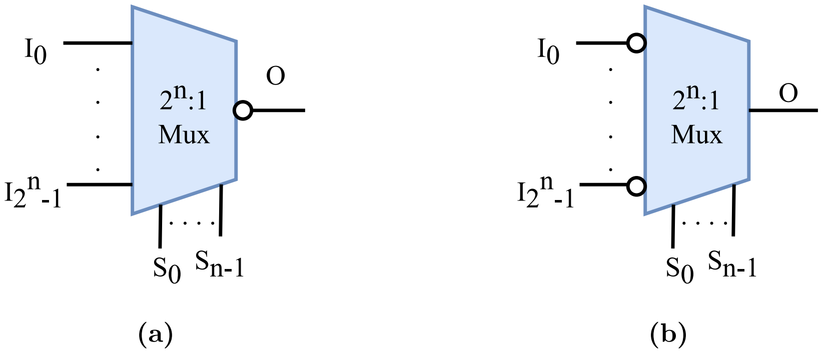


Fig. 7.9: *Bubble pushing on Multiplexer*

7.3.5 Register

A register may have multiple input sources and multiple output connections. The backward bubble-pushing approach for a register is as follows. When there is a bubble at each of the outputs of the register, it is pushed to all the inputs from the outputs. Otherwise, when there is at least one bubble at one of the outputs, we put the bubbles to all those outputs and inputs that do not have a bubble. An example of bubble pushing for an n-bit register with two inputs and three outputs is shown in Fig. 7.10. The original design in Fig. 7.10(a) is converted to the design in Fig. 7.10(b). Note that multiple inputs to a register are actually connected with Mux. However, for ease of representation, we have directly connected multiple inputs to the register in Fig. 7.10(a). Thus, the bubbles are actually pushed into the input of Muxes through the input of the register, i.e., from the output of the Muxes.

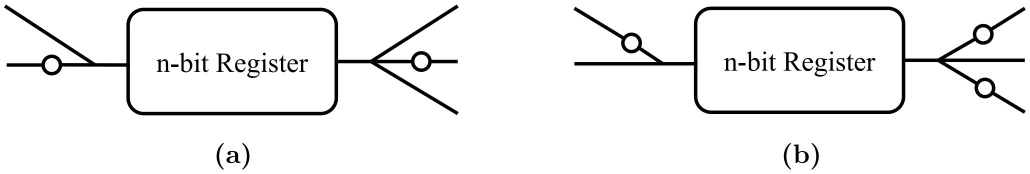


Fig. 7.10: *Bubble pushing on Register*

7.4 Proposed Defence to Protect Registers

We assume some of the inputs of the source code are sensitive/secret. The internal variables that depend explicitly or implicitly on the sensitive inputs are stored in the memory and/or registers in the hardware generated by HLS. However, the sensitive inputs are coming from a source (like tamper-proof memory) to which an attacker has no access. The overall flow of our proposed framework is given in Fig. 7.11.

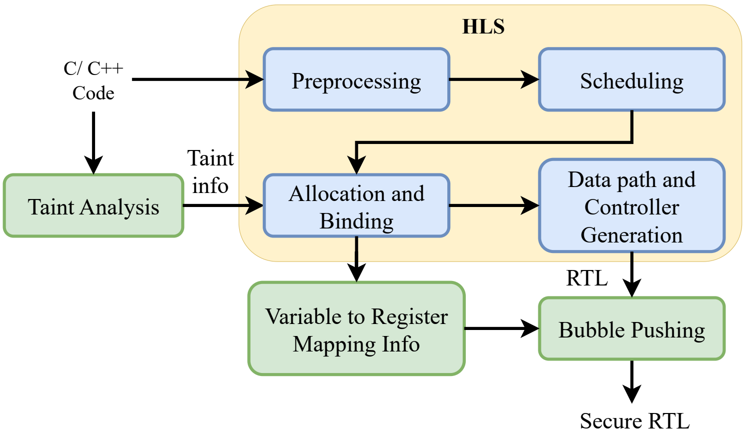


Fig. 7.11: *Overall Flow of Secure RTL Design*

The HLS flow takes the source C code as an input. A taint analysis based on our work proposed in Chapter 4 and Chapter 6 have been performed on the same C code to identify all the tainted or leaky variables for a given set of sensitive inputs of the C code. This taint info has been fed into the allocation and binding phase of the HLS flow to identify the sensitive registers by variable to register mapping information in the HLS. Our objective is to corrupt the sensitive information before storing it in the memory/register. To achieve that, we apply the bubble pushing on the generated RTL from HLS for the sensitive registers. The scan chain is inserted into the design later in the synthesis process. The tainted data is corrupted before being stored in the registers of the generated RTL. It ensures that even if the attacker obtains the tainted data through scan access, he or she gets the modified value of it. Since the direct correlation between the sensitive inputs and the register content is broken, meaningful side-channel analysis is difficult. The main contribution of this work is presented in the modules in green in Fig. 7.11.

The bubble pushing is presented as Algorithm 11. We obtain the list of sensitive registers from the mapping of tainted variables to registers during HLS. These sensitive registers are stored in a set R. The algorithm takes the HLS-generated RTL design and the set of sensitive registers R as inputs and generates the secure RTL design. We insert a random even number of bubbles at the output of each sensitive register in R. We ensure the bubbles

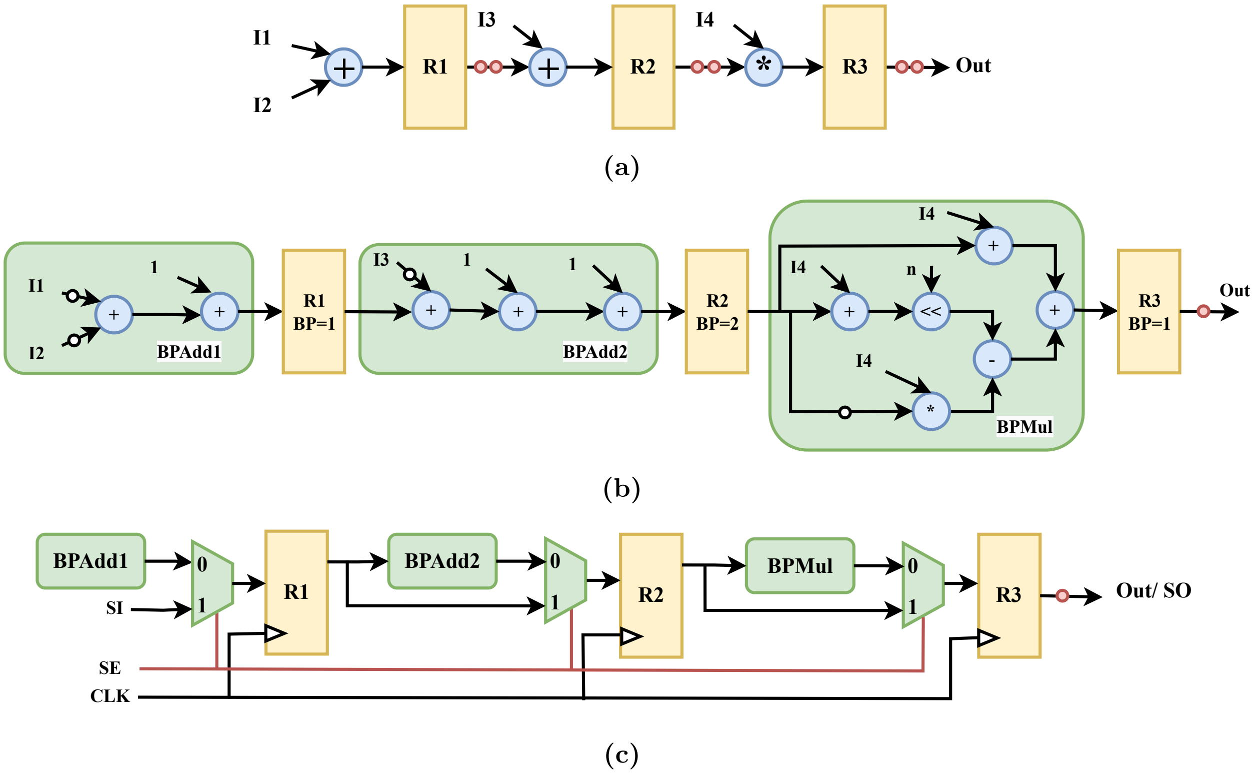
Algorithm 11: SecureReg(D, R)

```
Input: D: Original RTL design, R: set of sensitive registers
Output: Generated RTL design with protected sensitive registers
foreach sensitive register $r \in R$ do
    Insert random even number of bubbles at the output of $r$;
end
while the terminating condition is not true do
    Pick a set of inserted bubbles randomly in the design;
    Push the set of selected bubbles either in a forward or in a backward direction
    randomly up to one level;
    Update the modified components in the RTL design during the bubble pushing;
end
```

must be pushed in both forward and backward directions for a random number of iterations into the circuit to make sufficient corruption of the sensitive register values. The bubble pushing depends on the depth of the design from where the bubble is being inserted. We find the total fan-ins of the sensitive registers and set that as our terminating condition for the number of iterations of our bubble-pushing approach. The intuition is that a sensitive register with high fan-in has a more complex cone of influence (COI). Therefore, we need to push bubbles more number of times for such a register to make the corruption high. The time complexity of the proposed approach is in the order of $R * x$, where x is the highest depth of the bubble pushing performed for the set of sensitive registers R. *To the best of our knowledge, bubble pushing has not been explored at RTL design. Moreover, it has never been considered solely to corrupt the register content to protect them.*

7.4.1 Register Protection through Scan Access: An Example

An example of an RTL design with three registers R1, R2, and R3 is given in Fig. 7.12(a). Let us assume all these three registers are sensitive. Here, we have added two bubbles in red at the output of each register R1, R2, and R3. The generated RTL design after bubble pushing is presented in Fig. 7.12(b). We use a parameter *BP* (bubbles pushed) to keep track of the number of bubbles pushed through a register. We insert one bubble, two bubbles, and one bubble from the output of the sensitive registers R3 (BP=1), R2 (BP=2), and R1 (BP=1), respectively, in the backward direction. For simplicity, we have shown the example after inserting one/two bubbles. In general, this process of bubble pushing occurs



(a)

(b)

(c)

Fig. 7.12: *(a) A sample RTL Design; (b) Generated RTL Design after bubble pushing; (c) Generated RTL Design after scan chain insertion*

randomly and any even number of bubbles may be inserted. We have marked the modified components of the generated RTL design in Fig. 7.12(b) as BPAdd1, BPAdd2, and BPMul for the two adders and the multiplier after the bubble pushing, respectively. It may be noted that for the two adders in the original design in Fig. 7.12(a), the updated design in Fig. 7.12(b) is different due to the different number of bubbles pushed through the original adders.

The generated RTL design with the scan chain is given in Fig. 7.12(c). It is clear that in our approach, the secret information is modified in the circuit itself before passing to the scan chain due to the bubble pushing. Thus, the modified register value passed to the scan chain in red from the BPAdd1, BPAdd2, and BPMul. Here we are not corrupting the output values at the scan-out port SO. We are interested in protecting the intermediate secret values getting stored in the registers. Thus, we ensure the corruption of the functionality of the circuit for the intermediate sensitive registers without compromising the correct functionality at the output.

7.5 Experimental Results

7.5.1 Setup

The overall implementation of the proposed approach is a multi-step process. In the first step, we perform a taint analysis on the source specifications and obtain the tainted variables for all the sensitive inputs. We have used our implementation of taint-based quantification method in Chapter 6 for this purpose. Then, we obtain the sensitive registers based on the register-variable mappings and the RTL design using the Bambu HLS tool [61]. The set of these sensitive registers is fed as input to our implementation of the proposed bubble-pushing algorithm on the RTL design obtained from HLS. We have implemented the backward bubble-pushing approach on Verilog RTL. However, the forward bubble pushing can also be implemented in a similar fashion. In our implementation, we use PyVerilog [127] to obtain the abstract syntax tree (AST) from Verilog code. The bubble-pushing algorithm is implemented at the AST level. The RTL generation module generates the Verilog after bubble pushing from the modified AST.

Table 7.1: *Benchmark Characteristics*

Bench	#LOC	#LUT	#FF	#DSP	#BRAM	Freq. (MHz)
(1)	(2)	(3)	(4)	(5)	(6)	(7)
SHA	2,156	2,403	1,017	0	73	151.68
AES_ENC	4,763	761	791	0	5	240.21
AES_DEC	4,401	1,223	831	0	7	166.17
DES	1,773	159	133	0	0	505.56
MOTION	414	812	364	33	0	149.05
ARF	274	798	99	21	0	694.93
WAKA	282	1,375	66	0	0	723.07

We have considered four cryptographic algorithms (SHA, AES encoder, AES decoder, and DES) and three non-cryptographic computation-intensive benchmarks (MOTION, ARF, and WAKA) from Bambu distribution [61] in our experiments. The purpose of keeping non-cryptographic in our experiment is to analyze the performance overhead of our bubble-pushing algorithm on such behaviors. For the first set of benchmarks, the sensitive registers

Table 7.2: *Register content corruption value*

Bench (1)	#in (2)	#out (3)	#reg (4)	#high_reg (5)	#bubble_in (6)	cor_reg (%) (7)
SHA	7	11	121	4	8	100
AES_ENC	4	6	250	10	20	100
AES_DEC	4	8	206	10	20	100
DES	6	3	91	8	16	100
MOTION	13	10	87	5	10	100
ARF	16	9	28	4	8	100
WAKA	24	10	24	2	4	100

are identified using taint analysis. For the second set of benchmarks, an arbitrarily chosen subset of registers are considered sensitive for experiment purpose. We find the total fan-ins and fan-outs of these sensitive registers and set that as our terminating condition for the number of iterations of our bubble-pushing approach. The intuition is that the amount of bubble pushing is proportional to the register usage. In our experiments, the number of iterations is between 10 and 40 in Algorithm 11 for our benchmarks.

7.5.2 Performance Measures

We synthesize the HLS-generated RTL and the protected RTL in Xilinx ISE 14.7 to generate the performance report. We present the benchmark characteristics in Table 7.1. The characteristics of the HLS-generated RTLs are shown in terms of #LOC, #LUT, #FF, #DSP, #BRAM, and clock Frequency of the HLS-generated RTLs in the columns second to seventh, respectively. We present the total corruption of registers in Table 7.2. We present the number of inputs (#in), outputs (#out), and registers (#reg) for each benchmark in the second to the fourth column, respectively. From the set of total registers, we choose the set of high or sensitive registers based on the register-variable mappings using the Bambu HLS tool and present it in the fifth column (#high_reg). In our experiment, we have inserted two bubbles at the output of each sensitive register. The total number of bubbles inserted for each benchmark is presented in the sixth column (#bubble_in). Finally, we present the total percentage of register contents corrupted among the high or sensitive registers in the last column (cor_reg). Since we pushed bubbles across all sensitive registers, our proposed approach successfully protects all the sensitive registers.

7.5.3 Overhead Analysis

We report the area overhead of our bubble-pushed designs in Fig. 7.13. In our experiments, we compare the area overhead of the original RTL design and our generated RTL design. We find the area of a design based on the number of FFs, LUTs, DSPs, and BRAMs used by the design. We consider an abstract methodology to find the area by normalizing the weight of various components of FPGA from [79, 132]. The normalized values for FF, LUT, DSP, and BRAM are 1, 1, 238.5, and 115.4, respectively. It can be seen that the area overhead of the proposed design remains the same for most of the benchmarks. For the benchmark MOTION, the overhead is increased by 7.2%. We manually observed that MOTION has 14 multiplication operations, and there is a bubble pushing for 6 multiplications. As bubble pushing on multiplication has a higher overhead compared to other components, the area overhead is higher. However, we found that the average increase in area overhead is 3.86%.

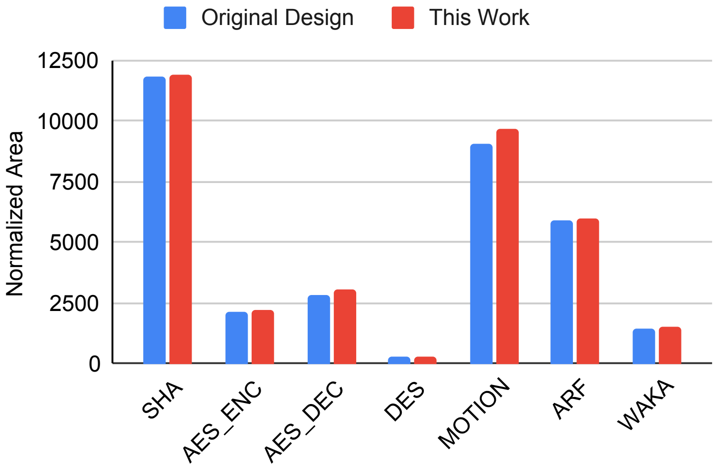


Fig. 7.13: *Normalized Area for Bubble pushing*

The timing overhead is reported in Fig. 7.14. It can be seen that the timing overhead is negligible for our designs except for MOTION. The number of LUTs in MOTION has increased to 1420 from 812 due to the bubble pushing on multipliers. Thus, the timing overhead in the case of MOTION is increased by 40%. However, the average increase in timing overhead is 4.58%. It can be observed that for the largest benchmarks like AES_ENC and AES_DEC, both the area and timing overhead are not very significant. The experimental results show that the proposed approach has negligible implementation overhead on the area

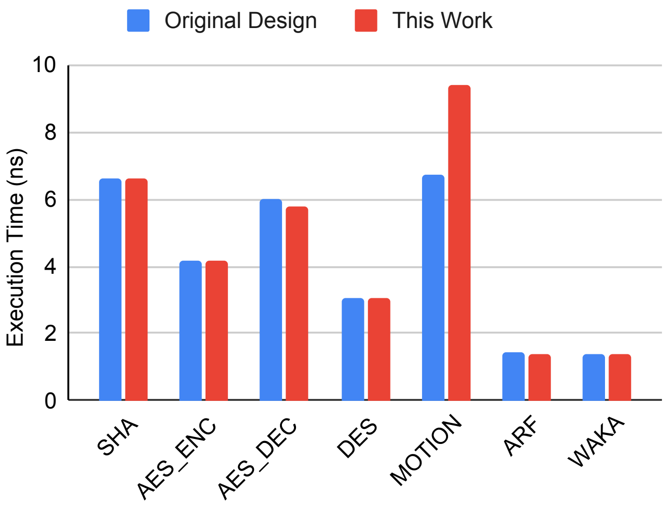


Fig. 7.14: *Execution Time for Bubble pushing*

and timing of the design.

7.6 A Case Study on AES

In order to check for the side-channel leakage of the protected design employing bubble-pushing countermeasure, power traces were recorded on Side-channel Attack Security Evaluation BOard (SASEBO) by passing numerous plaintexts to an AES design. SubBytes are the most significant step in the AES algorithm as they provide nonlinearity of a very high degree to the incoming data, which is performed using 16 substitution boxes (S-boxes). The choice of S-box is an important design step as SubBytes occupy the most area among the other round operations. Composite Field Arithmetic (CFA)-based S-boxes are gate-based designs that provide a very low area design offering the same functionality as a LUT-based S-box. Our work employs a similar type of S-box known as Canright's S-box [35].

Fig. 7.15 depicts the experimental setup where the board is connected to a power supply and an oscilloscope, whereas an application on a computer provides the plaintexts. The sampling rate of the oscilloscope is set to 1 GS/s, and the number of clocks required for AES processing is 12, with its design frequency chosen as 16 MHz (the highest permissible for the board), thus allowing 750 meaningful samples to be recorded per plaintext. Trigger events are captured by the board upon detecting which the AES operations are invoked.

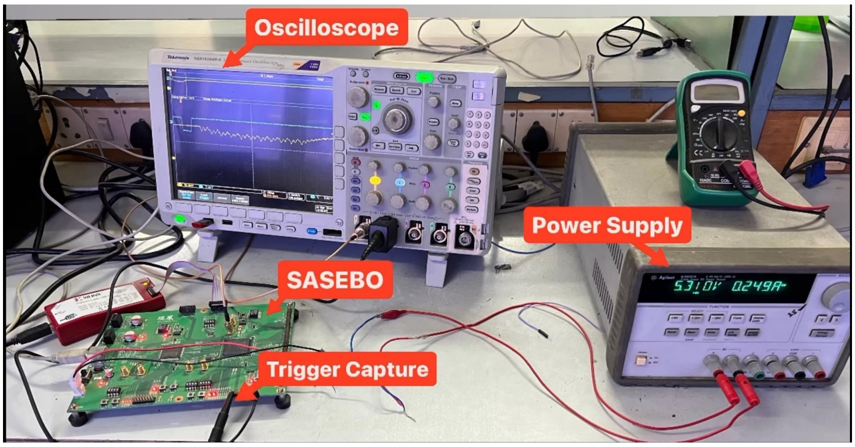


Fig. 7.15: *Experimental Setup for AES Case Study*

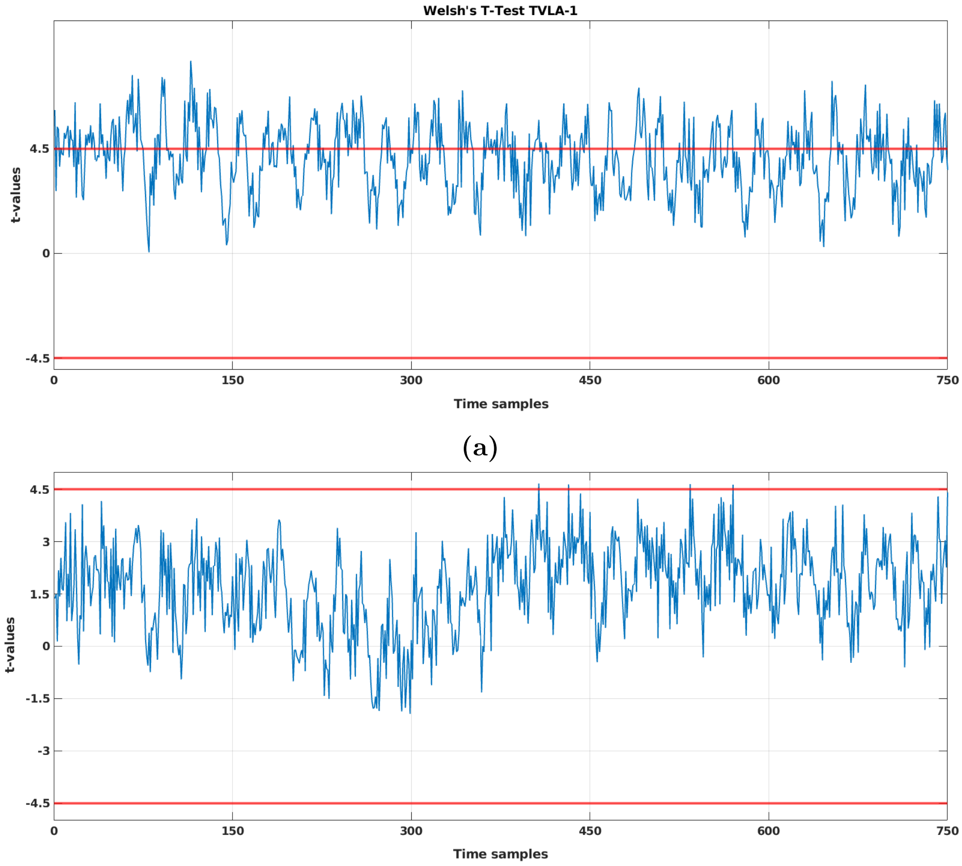
The cryptographic Field Programmable Gate Array (FPGA) on the board draws a current from the power supply, which is reflected on the oscilloscope as the AES power trace pattern. The trace is then sampled into various points, which are processed in MATLAB to provide the TVLA plots.

7.6.1 Discussion on TVLA

Test Vector Leakage Assessment (TVLA) is a general statistical tool used as a hardware security metric [63] to indicate the side-channel leakage of a cryptographic design under test [28,124]. It is used to analyze if two given sample sets originate from the same population source by analyzing their means. A large t-score obtained in the TVLA indicates the sets to have been extracted from the same population. The data set is partitioned into two sets, Q_0 and Q_1, with μ_0 (resp. μ_1), ρ_0 (resp. ρ_1), and n_0 (resp. n_1) representing the mean, standard deviation, and cardinality of set Q_0 (resp. Q_1). The t-test statistic, t, is then calculated as:

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{\rho_0^2}{n_0} + \frac{\rho_1^2}{n_1}}}$$

T-values of —±4.5— indicate a non-leaky design rendering it to be leakage-free with 99.99% confidence and secure from adversaries. The test involves the passage of plaintexts and calculation of the t-score as a part of TVLA. An unprotected AES design depicts t-scores outside the aforementioned threshold range, whereas the proposed protected design (with countermeasure) illustrates the scores within the threshold as shown in Fig. 7.16. Thus,



(a)

(b)

Fig. 7.16: *TVLA results comparison, (a) For unprotected AES (b) For protected AES*

our bubble-pushing algorithm protects the AES design from power side-channel attacks.

7.7 Discussion

In this section, we tried to address the following questions and their answers which may arise from the presented work and the proposed solution.

Q1: Why the taint analysis is performed at the C-level instead of at the RTL directly to identify the sensitive registers?

The sensitive registers can be found by performing a taint analysis at RTL itself. The existing RTL taint analysis approaches [67, 114] can be used for identifying all the sensitive registers for a chosen set of sensitive inputs. Instead, we prefer to perform the taint analysis at the source level and pass that information to RTL through the HLS. We notice that taint flow tracking at RTL does not capture all the dependencies of the sensitive inputs. The proposed methods in [67, 114] mainly focuses on finding the explicit dependencies of the sensitive inputs and ignore the implicit leaks due to the conditional blocks. This may lead to false negative scenarios, i.e., some registers may not be marked as sensitive during the taint analysis at RTL although these registers hold sensitive values in some instances. For example, in the following control block $if(i < h)\{a = b + 5; c = a - b\}else\{a = b + 5; c = a + b\}$, the leak of sensitive input h implicitly through the variable c and i due to the branch in the corresponding RTL design cannot be captured. The variable c has different symbolic values in the parallel paths of the control block, thus, it is leaking the sensitive input h in the condition. Moreover, no method exists in the literature that finds the fixed point of the leak in loops (Chapter 4) at RTL. For example, in the following loop $while(i < n)\{b = a; a = h; i++\}$, we cannot capture the leak of h through variable b due to the loop leak. Thus, the proposed approach in this thesis performs taint analysis at the source code to overcome these false negative scenarios due to implicit leaks and fixed points of loop leaks.

Q2. Isn't the C-level taint tracking lossy when it is translated into RTL?

The front-end of the HLS tool uses a C compiler like LLVM to optimize the input C code. The RTL is generated from this optimized C code by the back end of the HLS tool. The front end may introduce new leaks as shown in Chapters 5 and 6. Therefore, we should take the taint of the optimized IR obtained from the front-end compiler. It is shown that register allocation is secure without spilling and splitting in Chapter 3. Now if we show that there is no spilling and splitting in the generated RTL we can say that the taint tracking is not lossy. The variables in the optimized C code are in the SSA (Static Single Assignment) form. Thus, register splitting is invalid while generating the RTL from the optimized code. Moreover, no spilling is performed during HLS because all the operations are performed with registers, and memory is not used. It shows there is no spilling and splitting in the generated RTL. Therefore, the C-level taint tracking is not lossy.

Q3. Wouldn't bubble pushing impact the normal operation of the scan chain?

During the testing of the design, the designer would actually need to access the correct values. However, our proposed solution modifies the original values of the intermediate registers. The modified or updated values of the intermediate registers due to bubble pushing are shared with the test facility. This way, our proposed solution does not impact the normal operation of the scan chain.

Q4. Instead of changing the hardware, wouldn't it be more effective to actually protect scan chains to be read?

The existing techniques [78, 126] target protecting scan access by additional keys. However, such techniques are vulnerable to various attacks [89]. Moreover, the correct key is always embedded using a tamper-proof memory in a functional IC. Thus, few of the intermediate registers always have sensitive values. Therefore, various side-channel analysis can be performed on functional IC. However, in our proposed solution, the intermediate sensitive registers are always corrupted due to bubble pushing. Thus, altering the hardware component is effective, and performing the power side channel analysis is difficult. The real issue here is the leakage of sensitive information through the registers, which enables side-channel attacks. Hence, our goal is to corrupt the secret data that passes from the registers to the scan chain (without additional keys). In fact, the existing key-based scan protection techniques are complementary to our approach and may be added on top of our protection technique as double-layer protection.

7.8 Conclusion

This thesis proposes a novel approach to protect information leakage through the registers at the Register Transfer Level in which the register's content is corrupted using a bubble-pushing algorithm. Such a strategy breaks the statistical correlation between the register content and the sensitive inputs. In this thesis, we first introduce bubble-pushing for the various components at RTL. We then utilize it to protect the sensitive registers in the scan chain. The experimental results show that the overhead of the bubble-pushed designs is not significant. A case study of AES shows that our proposed technique does not leak power side-channel information, as validated by a TVLA experiment. It may be noted that the proposed solution targets the application-specific hardware generated through HLS.

8 Conclusions and Future Perspectives

This chapter summarizes the overall work in the thesis, highlights the contributions, and presents the possible future directions in the related domain.

8.1 Summary of Contributions

The contribution of the thesis is summarized as follows:

SRA: Secure Register Allocation for Trusted Code Generation: In Chapter 3, we analyzed the security issues of an essential optimization step of the compiler, i.e., register allocation. Initially, we analyzed the security of register allocation using the leaky triple concept of information leakage. However, due to its limitations, we again analyzed the information leakage using the notion of relative security definition. We have shown the register allocation is secure when there is no splitting and spilling. Then, we have shown that register allocation with splitting is as secure as the source program in our attack model. However, register allocation with spilling is not secure, and it introduces new leaky paths. In our experiment, we found that all the register allocations in the LLVM framework are leaky. We proposed a secure register allocation approach for the greedy register allocation in LLVM that aims to corrupt the memory content used for spilling after its last intended use. Moreover, the proposed approach has a negligible performance overhead, as shown in our experiment for a set of benchmarks.

QIL: Quantifying Information Leakage in a Program for Security Verification of Compiler Optimizations: Although taint analysis is a widely applied technique for security measurement in a program, it has the problem of either under-tainting or over-tainting. Therefore, a precise analysis of implicit flows is required for security verification. In Chapter 4, we proposed a quantification method for information leakage in a program using static taint analysis. We represent a program as an FSMD. We insert cutpoints into the program and obtain the path cover. The inserted cutpoints help us to analyze the implicit flows precisely and find the fixed point leak in loops. We have proposed a leak propagation vector to measure the leak in a program. We have also proposed methods to measure the implicit leaks precisely and to find the fixed point of information leakage due to loops. Finally, we have proposed three quantification parameters for information leakage in a program that aims to verify the relative security between the source and optimized programs. Our experimental study revealed that the SPARK compiler is actually leaky. The implementation in this work cannot handle arrays, functions, pointers, and dynamic memory allocations.

TVIL: Translation Validation of Information Leakage of Compiler Optimizations: Securing individual optimizations in a compiler is a challenging task as the compiler applies hundreds of optimizations. In Chapter 5, we proposed a TV method for security verification of compiler optimizations. Here, the applied optimizations and the order of optimizations are considered as a black box to generate the optimized program. We represented both the source and optimized programs as FSMDs. We have formally defined the relative security of two programs in terms of the path level relative security. Based on the formulation, we develop a translation validation method that works in a bi-simulation manner between two programs. It verifies the relative security between the source and optimized programs by propagating the leaks to the subsequent paths. To reduce the complexity of the method, we check certain look-ahead properties before making recursive calls. The experimental study found that the SPARK compiler is not relatively secure as it introduces new leaky paths.

MQIL: Model Checking based Quantification of Information Leakage in a Program: The state-of-the-art methods for quantification of information leakage are based on taint analysis. None of the existing works verified the relative security between the source and optimized programs as a concept of property-based testing using a model checker. In Chapter 6, we proposed a model checking based quantification method for information leakage in a program. The input program is modeled to verify the security property as assertions. The idea is to make two copies of the original program to verify the existence of any information flow from the sensitive inputs to any of the program variables and/or outputs. Our proposed approach is further applied to verify the relative security between the source and optimized programs. We have used the CBMC tool for quantifying the information leakage in the program. The CBMC can handle any constructs in a program, like, functions, arrays, pointers, etc. The total number of assertions failed by the CBMC is the total leakage of the program. Moreover, it generates a counter-example for each failed assertion. The experimental results revealed that the optimization phase of the LLVM compiler framework is not relatively secure for a set of cryptographic benchmarks.

SRIL: Securing Registers from Information Leakage at RTL: Scan chain is inserted into the design for testing purposes. However, it opens a gateway for attackers to access the sensitive contents stored in the registers through scan access. In Chapter 7, we proposed a method to secure the registers at RTL generated through HLS. The bubble-pushing approach is used in gate-level designs to modify the functional output of the design. In this work, we developed the bubble-pushing approach for various components of the RTL design and proposed a random bubble-pushing approach to corrupt the sensitive register contents. This gives the attacker no knowledge about the original secret data stored in the registers. The experimental results have shown that the proposed approach has a negligible performance overhead in terms of area and execution time. Moreover, it successfully corrupts all the sensitive registers without modifying the original functionality of the design. A case study on the AES benchmark has also been presented. It shows that the protected design is secure from the power side-channel attacks as validated by the TVLA experiment.

8.2 Future Directions

In this section, various possible future directions of the proposed methods are discussed.

8.2.1 Enhancement of Proposed Secure Register Allocation

Our proposed work in Chapter 3 can further be enhanced to secure the register allocation approach, as discussed below, in the context of the security of compiler optimizations.

- When an attacker has access to registers, splitting becomes leaky in a compiler, and sensitive register content needs to be flushed to ensure security.

- Spilling into cache instead of memory [65] for better performance in a compiler can potentially leak sensitive information through side-channel attacks. Thus, it is required to mitigate the leaks in the cache. One future direction of research is to explore how cache attacks [90,105] can exploit the leaks introduced by the compiler optimizations.
- To reduce the number of store zero operations for spilled memory locations in a secure register allocation approach, a taint analysis of the program after register allocation can be performed to identify only the tainted spills that require a store zero operation.

8.2.2 Counter-example Generation

Our quantification approach in Chapter 4 can be extended to report security flaws with a counter-example to the user. Our translation validation approach in Chapter 5 can also be extended to report a counter-example when our method finds that two programs are not relatively secure. Since the path level correspondence is available with the method and our methods locate the exact location of mismatch, counter-example generation may be a trivial enhancement as shown for program equivalence in [44].

8.2.3 Post-fixing of Leaks

In our proposed framework in Chapter 4, it is also possible to report the locations of the information leaks with additional bookkeeping. It would be interesting to explore if such information can be used to post-fix the leakage after the optimization phase of a compiler. It would be interesting to explore the identified security vulnerabilities in Chapter 5 post-fixed in order to achieve a secure compilation. The new leaks identified in our CBMC-based approach with counter-examples in Chapter 6 can be reported to the developer for practical benchmarks and further explore the possibility of post-fixing the information leaks introduced by the LLVM compiler framework in its optimization phase.

8.2.4 Security Verification of Optimization Phases

The relative security of individual compiler optimization phases such as -O1, -O2, -O3, and -O4 of compilers like LLVM and GCC can be evaluated with our tool in Chapter 6. Such analysis will help to identify which optimization level is most leaky. Moreover, the adaptability of our method in Chapter 5 for checking the security vulnerability of Just In

Time compilers [99,125] which apply various optimizations like reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common sub-expressions, etc., can further be explored. The optimizations in the LLVM compiler can be explored to develop a secure LLVM compiler. To achieve this, the first step is to identify the list of secure and insecure optimizations and stop the application of the insecure optimizations to generate the target code. This may need to sacrifice the performance to some extent at the cost of security.

8.2.5 Verification of Other Security Properties

This thesis verified the information flow security property of the program. However, other security properties (discussed in Section 2.1) like confidentiality, integrity, finite memory size, etc. should also be verified for a program to develop a secure compiler.

8.2.6 Bubble Pushing with Higher Corruption Rate

The bubble-pushing approach can further be extended to generate a secure RTL design with multiple data paths with different numbers of bubbles keeping the original functionality intact. The actual path will be selected random manner in runtime. This will lead to more uncertainty about the sensitive information in the design. The proposed work in Chapter 7 can also further be checked for the resilience of our countermeasure design against power analysis attacks using another security metric Measurements To Disclose.

8.3 Conclusion

This thesis presents methods to quantify the information leakage in a program using both taint analysis and property checking. Then it develops translation validation methods to verify the relative security between a source and its optimized program for well-known compilers like SPARK and LLVM. It also verifies the security of registers for general-purpose processors at the assembly level and for the application-specific hardware at the RTL. The impact of compiler optimizations on security is a little-explored area. This thesis addresses a few aspects in this direction. We show that SPARK and LLVM are actually leaky. We are confident that this thesis will stimulate further research in the compiler security domain.

Research Outcomes

From Thesis: Journals

1. **Priyanka Panigrahi**, Vemuri Sahithya, Chandan Karfa, and Prabhat Mishra, "Secure Register Allocation for Trusted Code Generation," in *IEEE Embedded Systems Letters* (ESL), vol. 14, no. 3, pp. 127-130, Sept. 2022, doi: 10.1109/LES.2022.3151096.

2. **Priyanka Panigrahi**, Abhik Paul and Chandan Karfa, "Quantifying Information Leakage for Security Verification of Compiler Optimizations," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), vol. 41, no. 11, pp. 4385-4396, Nov. 2022, doi: 10.1109/TCAD.2022.3200914.

3. **Priyanka Panigrahi**, and Chandan Karfa, "Translation Validation of Information Leakage of Compiler Optimizations", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (TCAD), vol. 42, no. 11, pp. 3585-3598, Nov. 2023, doi: 10.1109/TCAD.2023.3269954.

4. **Priyanka Panigrahi**, Karthik Maddala, Vignesh Ravichandra Rao, and Chandan Karfa, "Quantification of Information Leakage in a Program using Model Checker" (To be submitted).

From Thesis: Conferences

1. **Priyanka Panigrahi** and Chandan Karfa, "An Investigation into the Security of Register Allocation with Spilling and Splitting", in *IEEE Computer Society Annual Symposium on VLSI* (ISVLSI), Foz do Iguacu, Brazil, 2023, pp. 1-6, doi: 10.1109/ISVLSI59464.2023.10238662.

2. **Priyanka Panigrahi**, Vignesh Ravichandra Rao, Thockchom Birjit Singha, and Chandan Karfa, "SRIL: Securing Registers from Information Leakage at Register Transfer Level", in *37th International Conference on VLSI Design & 23rd International Conference on Embedded Systems* (VLSID), Kolkata, India, 2024, pp. 492-498, doi:10.1109/VLSID60093.2024.00088.

Outside Thesis

1. **Priyanka Panigrahi**, Rajesh Kumar Jha, and Chandan Karfa, (2019). "User Guided Register Manipulation in Digital Circuits", in *VLSI Design and Test* (VDAT), 2019, Communications in Computer and Information Science, vol 1066. Springer, Singapore. https://doi.org/10.1007/978-981-32-9767-8_39

Miscellaneous

1. **Priyanka Panigrahi**, "Security Verification of Compiler Optimizations: An Information Flow Perspective", accepted in *Ph.D. Forum of 37th International Conference on VLSI Design & 23rd International Conference on Embedded Systems* (VLSID), 2024.

2. **Priyanka Panigrahi**, "Quantification of information leakage and security verification of compiler optimizations", *Winner of Qualcomm Innovation Fellowship (QIF) India*, 2022.

3. **Priyanka Panigrahi**, "Security verification and quantification of information leakage for compiler transformations", accepted in *Ph.D. Forum of IEEE Women In Technology Conference* (WINTECHCON), 2022.

4. **Priyanka Panigrahi**, "Quantification of information leakage and security verification of compiler optimizations", *Finalist of Qualcomm Innovation Fellowship (QIF) India*, 2021.

References

[1] Benchmark source, https://github.com/ferrandi/panda-bambu, accessed May 05 2023. [Pg.135]

[2] Bounded model checking for software, https://www.cprover.org/cbmc/, accessed May 05 2023. [Pg.10], [Pg.135]

[3] CWE-14: Compiler Removal of Code to Clear Buffers, https://cwe.mitre.org/data/definitions/14.html, accessed March 05 2022. [Pg.21]

[4] CWE-733: Compiler Optimization Removal or Modification of Security-critical Code, https://cwe.mitre.org/data/definitions/733.html, accessed March 05 2022. [Pg.21]

[5] GCC Bug Report, https://gcc.gnu.org/bugs/. [Pg.4]

[6] LLVM Bug Report, https://bugs.llvm.org/. [Pg.4]

[7] LLVM Source, https://releases.llvm.org/download.html, accessed January 14 2022. [Pg.47]

[8] LLVM Test Suite, https://github.com/llvm/llvm-test-suite, accessed January 23 2023. [Pg.47]

[9] Parser for C language, https://github.com/eliben/pycparser, accessed May 05 2023. [Pg.134]

[10] A proactive approach to more secure code. `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`. [Pg.52]

[11] Translation of LLVM bitcode to C, https://github.com/staticafi/llvm2c, accessed May 05 2023. [Pg.15], [Pg.134]

[12] M. Abadi and G. Plotkin. On protection by layout randomization. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 337–351, July 2010. [Pg.19]

[13] C. Abate, R. Blanco, c. Ciobâcă, A. Durier, D. Garg, C. Hriţcu, M. Patrignani, E. Tanter, and J. Thibault. An extended account of trace-relating compiler correctness and secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(4), nov 2021. [Pg.22]

[14] M. Abderehman, T. Rakesh Reddy, and C. Karfa. Deeq: Data-driven end-to-end equivalence checking of high-level synthesis. In *ISQED*, pages 64–70, 2022. [Pg.9]

[15] J. Aerts and E. Marinissen. Scan chain design for test time reduction in core-based ics. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pages 448–457, 1998. [Pg.16]

[16] G. Agosta, A. Barenghi, and G. Pelosi. Compiler-based techniques to secure cryptographic embedded software against side-channel attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1550–1554, 2019. [Pg.52]

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. [Pg.1]

[18] L. Alrahis, M. Yasin, N. Limaye, H. Saleh, B. Mohammad, M. Al-Qutayri, and O. Sinanoglu. ScanSAT: Unlocking static and dynamic scan obfuscation. *IEEE Transactions on Emerging Topics in Computing*, 9(4):1867–1882, 2021. [Pg.32]

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. [Pg.7], [Pg.30]

[20] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA*, page 12–22, New York, NY, USA, 2011. Association for Computing Machinery. [Pg.29], [Pg.30]

[21] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, dec 1994. [Pg.90]

[22] K. Banerjee, C. Karfa, D. Sarkar, and C. A. Mandal. Verification of code motion techniques using value propagation. *IEEE TCAD*, 33(8):1180–1193, 2014. [Pg.5], [Pg.81], [Pg.118]

[23] K. Banerjee, D. Sarkar, and C. Mandal. Extending the fsmd framework for validating code motions of array-handling programs. *IEEE TCAD*, 33(12):2015–2019, 2014. [Pg.86]

[24] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *ISSTA*, page 13–24, New York, NY, USA, 2010. Association for Computing Machinery. [Pg.29]

[25] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 2–15, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. [Pg.26]

[26] K. Basu, D. Soni, M. Nabeel, and R. Karri. Nist post-quantum cryptography- a hardware evaluation study. Cryptology ePrint Archive, Paper 2019/047, 2019. `https://eprint.iacr.org/2019/047`. [Pg.140]

[27] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations.* MITRE Technical Report 2547, Volume I, 1973. [Pg.7], [Pg.8], [Pg.9], [Pg.27], [Pg.31]

[28] D. Bellizia, S. Bongiovanni, P. Monsurrò, G. Scotti, A. Trifiletti, and F. B. Trotta. Secure double rate registers as an rtl countermeasure against power analysis attacks. *IEEE transactions on very large scale integration (vlsi) systems*, 26(7):1368–1376, 2018. [Pg.155]

[29] F. Besson, A. Dang, and T. Jensen. Securing compilation against memory probing. In *PLAS*, pages 29–40, 2018. [Pg.22], [Pg.23], [Pg.35], [Pg.51]

[30] F. Besson, A. Dang, and T. Jensen. Information-Flow Preservation in Compiler Optimisations. In *CSF*, pages 1–13, Hoboken, United States, June 2019. IEEE. [Pg.22], [Pg.23], [Pg.24], [Pg.25], [Pg.140]

[31] D. Binkley, N. Gold, and M. Harman. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.*, 16(2):8–39, Apr. 2007. [Pg.6]

[32] J. Bonneau and I. Mironov. Cache-collision timing attacks against aes. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 201–215, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. [Pg.31]

[33] A. Bosu, F. Liu, D. D. Yao, and G. Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 71–85, New York, NY, USA, 2017. Association for Computing Machinery. [Pg.7]

[34] D. Boxler. Static taint analysis tools to detect information flows. 2018. [Pg.30]

[35] D. Canright et al. A very compact s-box for aes. In *Ches*, volume 3659, pages 441–455. Springer, 2005. [Pg.154]

[36] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Krügel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015. [Pg.30]

[37] D. Ceara, L. Mounier, and M.-L. Potet. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 371–380, 2010. [Pg.7], [Pg.28], [Pg.29], [Pg.30], [Pg.56], [Pg.123]

[38] G. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982. [Pg.8], [Pg.34]

[39] G. Chaitin et al. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, Jan. 1981. [Pg.8], [Pg.34]

[40] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), feb 2020. [Pg.4]

[41] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. *SIGPLAN Not.*, 48(6):197–208, June 2013. [Pg.5], [Pg.21]

[42] B. Chess and G. McGraw. Static analysis for security. *IEEE Security and Privacy*, 2:76–79, 2004. [Pg.6]

[43] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007. [Pg.6]

[44] R. Chouksey, C. Karfa, K. Banerjee, P. K. Kalita, and P. Bhaduri. Counter-example generation procedure for path-based equivalence checkers. *IET Softw.*, 13(4):280–285, 2019. [Pg.162]

[45] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *STTT*, 2:410–425, 2000. [Pg.74], [Pg.113]

[46] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking, 1st Edition*. MIT Press, 2001. [Pg.4], [Pg.74]

[47] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, page 196–206, New York, NY, USA, 2007. Association for Computing Machinery. [Pg.7], [Pg.29]

[48] K. D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In K. Koskimies, editor, *Compiler Construction*, pages 174–187, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. [Pg.8]

[49] A. Cui, Y. Luo, H. Li, and G. Qu. Why current secure scan designs fail and how to fix them? *Integration*, 56:105–114, 2017. [Pg.31], [Pg.32]

[50] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang. Testing static analyzers with randomly generated programs. In A. E. Goodloe and S. Person, editors, *NASA Formal Methods*, pages 120–125, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. [Pg.5], [Pg.21]

[51] L. S. de Araújo, L. A. Marzulo, T. A. Alves, F. França, I. Koren, and S. Kundu. Building a portable deeply-nested implicit information flow tracking. *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020. [Pg.29]

[52] C. Deng and K. S. Namjoshi. Securing a compiler transformation. In X. Rival, editor, *Static Analysis*, pages 170–188, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. [Pg.8], [Pg.9], [Pg.23], [Pg.26], [Pg.31], [Pg.57], [Pg.79], [Pg.115]

[53] C. Deng and K. S. Namjoshi. Securing the ssa transform. In F. Ranzato, editor, *Static Analysis*, pages 88–105, Cham, 2017. Springer International Publishing. [Pg.8], [Pg.9], [Pg.24], [Pg.31]

[54] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. [Pg.7], [Pg.8], [Pg.9], [Pg.27], [Pg.31], [Pg.115]

[55] F. Derakhshan, Z. Zhang, A. Vasudevan, and L. Jia. Towards end-to-end verified tees via verified interface conformance and certified compilers. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 324–339, 2023. [Pg.22]

[56] V. D'Silva et al. The correctness-security gap in compiler optimization. In *IEEE Security and Privacy Workshops*, pages 73–87, 2015. [Pg.5], [Pg.8], [Pg.24], [Pg.57], [Pg.79]

[57] A. El-Korashy, R. Blanco, J. Thibault, A. Durier, D. Garg, and C. Hriţcu. Secureptrs: Proving secure compilation with data-flow back-translation and turn-taking simulation. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*, pages 64–79, 2022. [Pg.22]

[58] A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. Capableptrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–16, 2021. [Pg.22]

[59] A. Fanti, C. C. Perez, R. Denis-Courmont, G. Roascio, and J.-E. Ekberg. Towards register spilling security using llvm and arm pointer authentication. *IEEE TCAD*, pages 1–1, 2022. [Pg.24], [Pg.25], [Pg.140]

[60] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, page 576–587, New York, NY, USA, 2014. Association for Computing Machinery. [Pg.7]

[61] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE, Dec 2021. [Pg.151]

[62] R. W. Floyd. Assigning meanings to programs. 1993. [Pg.61]

[63] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011. [Pg.155]

[64] K. Gondi, P. Bisht, P. Venkatachari, A. P. Sistla, and V. N. Venkatakrishnan. Swipe: Eager erasure of sensitive data in large scale systems software. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, page 295–306, New York, NY, USA, 2012. Association for Computing Machinery. [Pg.7], [Pg.25], [Pg.51]

[65] J. R. Goodman and W. C. Hsu. On the use of registers vs. cache to minimize memory traffic. *SIGARCH Comput. Archit. News*, 14(2):375–383, may 1986. [Pg.162]

[66] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015. [Pg.7], [Pg.30]

[67] X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin. Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 91–100, 2019. [Pg.156], [Pg.157]

[68] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design*, pages 461–466, 2003. [Pg.14], [Pg.81], [Pg.118]

[69] S. Gupta, A. Rose, and S. Bansal. Counterexample-guided correlation algorithm for translation validation. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. [Pg.4], [Pg.9]

[70] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. [Pg.61]

[71] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. SUPOR: Precise and scalable sensitive user input detection for android apps. In *USENIX Security*, pages 977–992, Washington, D.C., Aug. 2015. USENIX Association. [Pg.7]

[72] Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481. Springer, 2003. [Pg.140]

[73] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely. Local memory via layout randomization. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 161–174, June 2011. [Pg.19]

[74] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *IEEE S&P*, pages 6 pp.–263, 2006. [Pg.7]

[75] M. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011. [Pg.7], [Pg.29]

[76] C. Karfa, C. A. Mandal, and D. Sarkar. Formal verification of code motion techniques using data-flow-driven equivalence checking. *ACM TODAES*, 17(3):30:1–30:37, 2012. [Pg.5], [Pg.58], [Pg.59]

[77] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar. An equivalence-checking method for scheduling verification in high-level synthesis. *IEEE TCAD*, 27(3):556–569, 2008. [Pg.5], [Pg.9], [Pg.60]

[78] R. Karmakar, S. Chattopadhyay, and R. Kapur. A scan obfuscation guided design-for-security approach for sequential circuits. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 67(3):546–550, 2020. [Pg.32], [Pg.158]

[79] T. A. Khader, N. Sarma, and C. Karfa. Imagespec: Efficient high-level synthesis of image processing applications. In *Euromicro Conference on Digital Systems Design 2022 (DSD22)*. IEEE, 2022. [Pg.153]

[80] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. [Pg.31]

[81] V. K. Kurhe, P. Karia, S. Gupta, A. Rose, and S. Bansal. Automatic generation of debug headers through blackbox equivalence checking. In *CGO*, CGO '22, page 144–154. IEEE Press, 2022. [Pg.4], [Pg.9]

[82] J. Lee, M. Tehranipoor, C. Patel, and J. Plusquellic. Securing designs against scan-based side-channel attacks. *IEEE Transactions on Dependable and Secure Computing*, 4(4):325–336, 2007. [Pg.32]

[83] S. Lerner, T. D. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2003. [Pg.4]

[84] X. Leroy. Formal verification of an optimizing compiler. In *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*, pages 25–25, 2007. [Pg.4]

[85] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009. [Pg.4]

[86] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, July 2008. [Pg.4]

[87] A. Leung, D. Bounov, and S. Lerner. C-to-verilog translation validation. In *MEMOCODE*, pages 42–47, 2015. [Pg.9]

[88] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, volume 1, pages 280–291, 2015. [Pg.30]

[89] N. Limaye and O. Sinanoglu. Dynunlock: Unlocking scan chains obfuscated using dynamic keys. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 270–273, 2020. [Pg.31], [Pg.32], [Pg.158]

[90] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015. [Pg.162]

[91] Y. Liu and A. Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In R. Capilla, R. Ferenc, and J. C. Dueñas, editors, *CSMR*, pages 146–155. IEEE Computer Society, 2010. [Pg.7], [Pg.30], [Pg.123]

[92] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012. [Pg.7]

[93] Y. Lyu and P. Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018. [Pg.52]

[94] R. S. Meurer, T. R. Mück, and A. A. Fröhlich. An implementation of the aes cipher using hls. In *2013 III Brazilian Symposium on Computing Systems Engineering*, pages 113–118, 2013. [Pg.140]

[95] G. D. Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Higher Education, 1st edition, 1994. [Pg.11]

[96] P. Mishra, R. Morad, A. Ziv, and S. Ray. Post-silicon validation in the soc era: A tutorial introduction. *IEEE Design Test*, 34(3):68–92, 2017. [Pg.51]

[97] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In D. H. Won and S. Kim, editors, *ICISC*, pages 156–168, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. [Pg.21]

[98] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann publishers Inc., San Francisco, CA, USA, 1998. [Pg.1], [Pg.61]

[99] M. O. Myreen. Verified just-in-time compiler on x86. *SIGPLAN Not.*, 45(1):107–118, jan 2010. [Pg.163]

[100] K. S. Namjoshi and L. M. Tabajara. Witnessing secure compilation. In *VMCAI*, page 1–22, Berlin, Heidelberg, 2020. Springer-Verlag. [Pg.22], [Pg.23], [Pg.115]

[101] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. UIPicker: User-Input privacy identification in mobile applications. In *USENIX Security*, pages 993–1008, Washington, D.C., Aug. 2015. USENIX Association. [Pg.7]

[102] V. K. Nandivada, F. M. Q. Pereira, and J. Palsberg. A framework for end-to-end verification and evaluation of register allocators. In H. R. Nielson and G. Filé, editors, *Static Analysis*, pages 153–169, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. [Pg.4], [Pg.41]

[103] S. Narayanan and M. Breuer. Reconfiguration techniques for a single scan chain. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(6):750–765, 1995. [Pg.16]

[104] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. *SIGPLAN Not.*, 33(5):333–344, may 1998. [Pg.4]

[105] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. [Pg.162]

[106] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, Apr. 2015. [Pg.20], [Pg.22], [Pg.23]

[107] M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, Feb. 2019. [Pg.5]

[108] M. Patrignani and D. Garg. Secure compilation and hyperproperty preservation. In *IEEE 30th CSF'17*, pages 392–404, Aug 2017. [Pg.8]

[109] M. Patrignani and D. Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1), feb 2021. [Pg.22], [Pg.23]

[110] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):798–808, 2019. [Pg.140]

[111] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, page 151–166, Berlin, Heidelberg, 1998. Springer-Verlag. [Pg.4], [Pg.5]

[112] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, Sept. 1999. [Pg.8]

[113] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *PLDI*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery. [Pg.21]

[114] L. M. Reimann, L. Hanel, D. Sisejkovic, F. Merchant, and R. Leupers. Qflow: Quantitative information flow for security-aware hardware design in verilog. In *IEEE International Conference on Computer Design*, 2021. [Pg.156], [Pg.157]

[115] S. Rideau and X. Leroy. Validating register allocation and spilling. In *Compiler Construction*, pages 224–243, 2010. [Pg.41]

[116] J. Rosemann, S. Schneider, and S. Hack. Verified spilling and translation validation with repair. In *ICITP'10*, pages 427–443, 2017. [Pg.41]

[117] R. Sadhukhan, S. Saha, and D. Mukhopadhyay. Shortest path to secured hardware: Domain oriented masking with high-level-synthesis. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 223–228, 2021. [Pg.140]

[118] Y. Sao and S. S. Ali. Security analysis of state-of-the-art scan obfuscation technique. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 599–602, 2021. [Pg.31]

[119] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *Euro S&P*, pages 15–30, 2016. [Pg.30]

[120] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, pages 317–331, 2010. [Pg.7], [Pg.29]

[121] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, sep 2005. [Pg.100]

[122] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. *SIGPLAN Not.*, 48(10):391–406, oct 2013. [Pg.9], [Pg.100]

[123] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '13, page 391–406, New York, NY, USA, 2013. Association for Computing Machinery. [Pg.100]

[124] A. Singh, M. Kar, V. C. K. Chekuri, S. K. Mathew, A. Rajan, V. De, and S. Mukhopadhyay. Enhanced power and electromagnetic sca resistance of encryption engines via a security-aware integrated all-digital ldo. *IEEE Journal of Solid-State Circuits*, 55(2):478–493, 2019. [Pg.155]

[125] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000. [Pg.163]

[126] M. Taherifard, H. Beitollahi, F. Jamali, A. Norollah, and A. Patooghy. Mist-scan: A secure scan chain architecture to resist scan-based attacks in cryptographic chips. In *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pages 135–140, 2020. [Pg.32], [Pg.158]

[127] S. Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015. [Pg.151]

[128] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015. [Pg.7]

[129] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC*, pages 289–298, 2008. [Pg.30]

[130] X. Wang, H. Ma, and L. Jing. A dynamic marking method for implicit information flow in dynamic taint analysis. In *SIN*, page 275–282, New York, NY, USA, 2015. Association for Computing Machinery. [Pg.29]

[131] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, SOSP '13, page 260–275, New York, NY, USA, 2013. Association for Computing Machinery. [Pg.21]

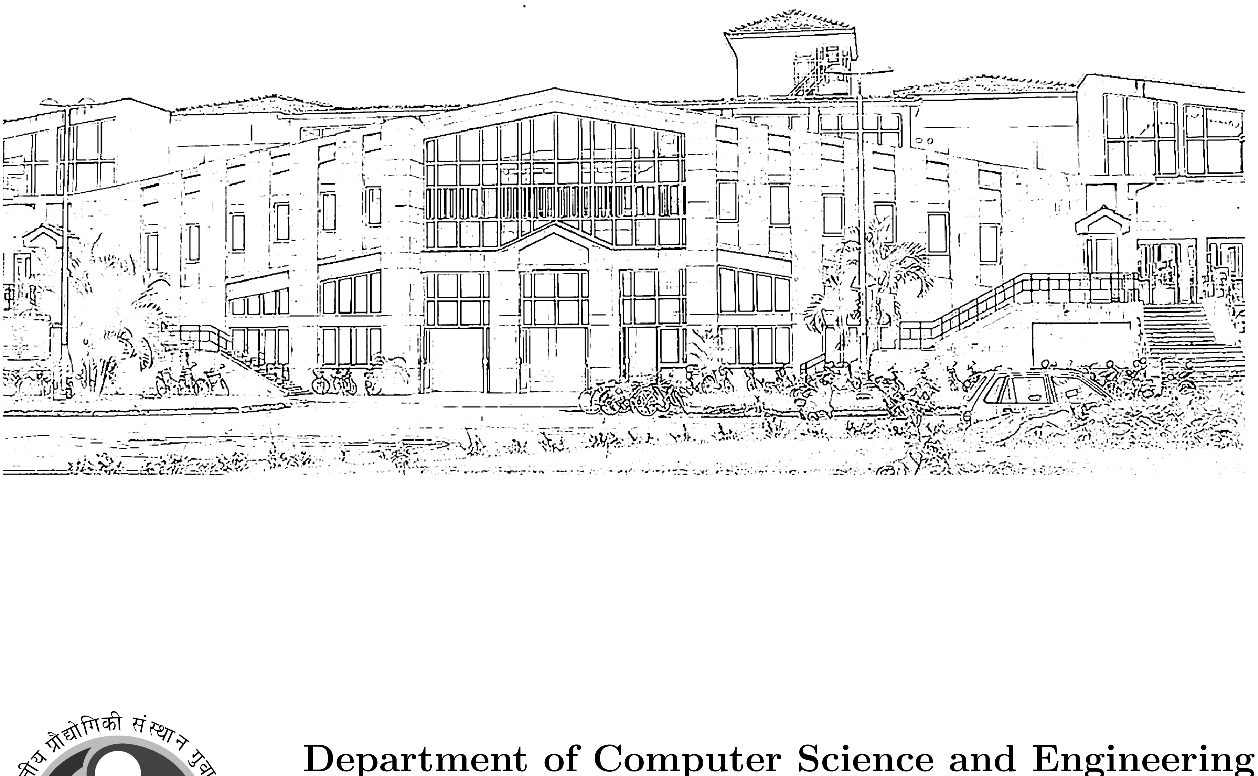
[132] H. Wong, V. Betz, and J. Rose. Comparing fpga vs. custom cmos and the impact on processor microarchitecture. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 5–14, New York, NY, USA, 2011. Association for Computing Machinery. [Pg.153]

[133] B. Yang, K. Wu, and R. Karri. Scan based side channel attack on dedicated hardware implementations of data encryption standard. In *2004 International Conferce on Test*, pages 339–344, 2004. [Pg.139]

[134] M.-K. Yoon, N. Salajegheh, Y. Chen, and M. Christodorescu. Pift: Predictive information-flow tracking. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, page 713–725, New York, NY, USA, 2016. Association for Computing Machinery. [Pg.29]

[135] R. Zhang, S. Huang, Z. Qi, and H. Guan. Static program analysis assisted dynamic taint tracking for software vulnerability discovery. *Computers Mathematics with Applications*, 63(2):469–480, 2012. Advances in context, cognitive, and secure computing. [Pg.29]

[136] X. Zhang, X. Wang, R. Slavin, and J. Niu. Condysta: Context-aware dynamic supplement to static taint analysis. In *IEEE S&P*, pages 796–812, 2021. [Pg.30]



Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039, India