A TOOL FOR VERIFYING NONINTERFERENCE PROPERTY FOR HDL

by
Doğuhan Gümüşoğlu

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2017

A TOOL FOR VERIFYING NONINTERFERENCE PROPERTY FOR HDL

APPROVED BY:

Assist. Prof. Dr. Onur Demir
(Thesis Supervisor) .... [illegible]

Assoc. Prof. Dr. Fatih Uğurdağ .... [illegible]

Assist. Prof. Dr. Tacha Şerif .... [illegible]

DATE OF APPROVAL: ..../..../2017

ACKNOWLEDGEMENTS

It is with immense gratitude that I acknowledge the support and help of my Professor Onur Demir. Pursuing my thesis under his supervision has been an experience which broadens the mind and presents an unlimited source of learning.

I would also like to acknowledge Professor Jakub Szefer of the Electrical Engineering & Computer Science at Yale University as the second reader of this thesis, and I am gratefully indebted to him for his guidance throughout the research.

I thank Research Assistants Yılmaz Serhan Gener, Ph.D. candidate Shuwen Deng, and Ph.D. candidate Wenjie Xiong.

Finally, I would like to thank my family and my wife Fulya for their endless love and support, which makes everything more beautiful.

ABSTRACT

A TOOL FOR VERIFYING NONINTERFERENCE PROPERTY FOR HDL

Hardware description languages started to adopt high-level functionalities. Through the flexibility of high-level programming, hardware designers can leverage object-oriented programming and functional programming paradigms to specify the hardware for quicker design time. This transition to high-level hardware descriptions created a new set of problems about verifying their noninterference property. By extending a high-level HDL, this thesis shows how to approach and solve this new set of problems and thus preventing information leakage. SecChisel (based on Chisel HDL) gives the ability to design circuits which have algorithmically verifiable noninterference property. SecChisel handles the creation of temporary variables and cascaded propagation of security tags to these new variables which are unique problems that only occurs when the host language creates an intermediate representation of the designed circuit. SecChisel algorithmically verifies a design by transforming and modeling the design in a satisfiability modulo theory (SMT) solver. By building and verifying hardware designs, we demonstrate that SecChisel provides a simple way to verify circuit design's noninterference property at compile time with no overhead.

ÖZET

DONANIM TASARLAMA DİLİNİN BİRBİRİNE KARIŞMAMIŞLIK ÖZELLİĞİNİ DOĞRULAYAN ARAÇ

Donanım tanımlama dilleri giderek daha yüksek seviye özellikleri bünyelerinde barındırmaya başladı. Yüksek seviye programlamanın getirdiği esneklik ile donanım tasarımcıları obje tabanlı programlama ve fonksiyonel programlama gibi programlama paradigmalarından faydalanıp, dizayn sürelerini kısalttılar. Donanım tasarımlarının yüksek seviyeye geçmesi bu tasarımların veri akışı güvenliklerinin doğrulanmasında bir takım yeni sorunlar üretti. Bu çalışma yüksek seviye bir donanım tanımlama dilini genişleterek, ortaya çıkan bu yeni sorunların nasıl çözüleceğini ve bu sayede tasarımlarda veri sızmasının nasıl önüne geçilebileceğini göstermektedir. SecChisel (Chisel üzerine temellendirilmiş) veri akışı güvenliği özelliklerinden birbirine karışmamışlık (noninterference) özelliği kanıtlanabilir devreler tasarlanmasına imkan sunar. SecChisel ara değişken yaratılımı ve bu değişkenlere kademeli olarak aktarılması gerekilen güvenlik etiketleri gibi sorunları çözer. Bu sorunlar sadece kullanılan ana dilin direkt bir devre sentezlemesi yerine başka bir orta seviye dile çıktı vermesine özel durumlardır. SecChisel'ın devre veri akışı güvenliğini doğrulama yöntemi, devreyi bir takım dönüştürmelerden geçirip SMT çözücü formatında tekrar modellemektir. SecChisel'ın veri akışı güvenliğini derleme süresinde ve her hangi bir ek yük getirmeden doğrulayabildiği dizayn edip test etmiş olduğumuz devreler ile kanıtlanmıştır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

$\vee$	Lattice join operation
$\wedge$	Lattice meet operation
HDL	Hardware description language
IR	Intermediate representation
LEQ	Less or equal
RTL	Register-transfer level
SMT	Satisfiability modulo theories

1. INTRODUCTION

Verifying a hardware is crucial and it is recognized as the largest task in silicon development. Hardware verification for correctness (functional) is checking if a design correctly implements its specification. There is also hardware verification for security properties which is the process of checking that the design has intended information flow. Information flow control is as important as the correctness of the design since it offers a convenient way to prevent unwanted exposure and modification of information in systems. Information flow policies are successfully enforced at language and operating system levels, but without verifying that the underlying hardware system is not violating any information flow policy rule by its design, the software that built upon it can never be secure.

Security of a design is depended on both hardware and software. The correctness of the protections provided by these layers as a whole makes the security of the design. This creates a need to verify the correctness of security provided by both software and hardware components. Insufficient verification at design time may lead to vulnerabilities in the system. Hardware verification is especially more crucial in this sense since, after fabrication, it is almost impossible to patch the vulnerability, unlike software.

State of the art security verification approaches illustrated their limitations to catch bugs at design time. For example, Core 2 Duo processor family of Intel has 129 bugs [1] and a substantial amount of them are security related [2]. This kind of bugs motivated researchers to develop new methods for verification of security properties of such systems at design time.

Model checking verification methods proves a system for its properties by modeling a state of the system. The modelled system then becomes a set of states with a set of transitions between these states. Model checking tools can then be used to verify that desired properties exist in the system. The algorithmic approach in this sense refers to the techniques for state space explorations. The state exploration techniques are more general and convenient than their simulation based counterpart. Although formal verification methods were being used for functional verification of both hardware and software for a long time, using the formal method for verifying security properties is relatively new. Our work aims to offer a formal verification procedure for verifying a high-level HDL's noninterference [19, 17] property.

An alternative approach is using a theorem prover rather than a model checker. While the proof-based approach has several advantages such as handling complex systems because it does not navigate through a system state space, it also has its own drawbacks. One of such is, transforming a complex design to a theorem equivalence.

New hardware description languages started to adopt high-level features. By utilizing object-oriented programming and functional programming paradigms they provide a more expressive environment for designers.

One such language is Chisel [3]. It is being developed by Berkeley University. Chisel is embedded in the high-level programming language Scala [4]. Verifying noninterference property of Chisel requires a different approach to the problem due to the workflow of the language. While it is possible to synthesis directly from Verilog-like languages, Chisel generates an equivalent intermediate representation of the design named FIRRTL (Flexible Intermediate Representation for RTL) [18]. This intermediate representation is not only for Chisel and can be targeted by other languages as well. The intermediate language FIRRTL is stripped out of the high-level programming concepts and it is much closer to the Verilog code. FIRRTL representation of the circuit can then be converted to various other formats, one of which is Verilog.

However, transformation to the FIRRTL generates variables which were not defined by the designer. This creates the problem of handling such variables without designer's knowledge while verifying the design for noninterference property. Such problems do not exist in Verilog-like languages because no other intermediate variables are created in the process and everything is available to the designer directly. This thesis aims to solve problems emerged from using a high-level HDL and verify its noninterference property. Our approach solves the intermediate variable problem at compile time and without adding any overhead to the design.

The major contributions of this work are:

- An algorithmic approach to verify noninterference property of HDL designs.
- Providing tracking of information flow for designs by extending Chisel and creating a new HDL SecChisel.
- Security tag assignments to intermediate variables which are not available to the designer.
- Static and dynamic security tags which allow designer to create information flow policies at any complexity.
- Providing a convenient way to specify information flow policies with lattices and transformation of these lattice structures to different language domains.
- Verifying the design as a whole. Input and output ports of inner modules are also traced for noninterference.

1.1. SECURITY PROPERTIES

For a system to be secure, it must explicitly provide security guarantees and a Trusted Computing Base (TCB) which those security guarantees are built upon. TCB is comprised of both hardware and software layers. These multiple layers work together and need to be verified in order to label the whole system as secure. Security properties are explained below.

1.1.1. Noninterference

Noninterference is an analysis on the interaction between low-security and high-security entities. Interference between low and high-security entities is analyzed to confirm that low-security entities can not observe any difference in the system based on high-security entities or nonsensitive inputs. It is allowed for a high-security entity to observe any difference in the system or in the low-security entity.

Noninterference means that high-security information is not leaking and thus observable by low-security (insecure) entities. Confidentiality can be verified through noninterference when the system is partitioned into entities such as low/high.

1.1.2. Confidentiality, Integrity, and Availability

Confidentiality, integrity, and availability are the three major security properties when analyzing a system [5].

These properties can be defined as follows:

- "Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities" [6]
- "Integrity is the prevention of unauthorized modification of protected information without detection" [6]
- "Availability is the provision of services and systems to legitimate users when requested or needed" [6]

A system verified on security property assumes that reliability (protection from random system faults) is already provided by the system. The whole focus is on an attack by a smart adversary, given that the system works as expected.

Among three properties, availability is not often considered by verification systems to verify, since it is difficult to prove. An attacker can cut the power off or disable network connections, or use some other methods to make the system unreachable. Having said that, if a system is not available, it does not provide any kind of information to the attacker as well nor does it allow any information provided by the system to be modified. The system is just unusable. Thus, availability will be out of our scope of research.

1.1.3. Information Flow

Information flow is about the transfer of information between different entities. We can define it as the information transfer from variable x to another variable y in a process. Though not all flows are permitted since the system is not allowed to leak secret information to public observers.

The information flow can be explicit. Such as *x = y* where the information of y is moved to *x*. Information flow can also be implicit such as, *x = 0; if(y) { x = 1; }*. The value of *x* gives a hint on whether the y is true or not, even though there was never an assignment between *x* and *y*.

A system can enforce properties such as "there is no implicit or explicit information flow between *x* and *y*". Only the verification of these properties would mean verification of the system of its secure information flow property. Information flow happens through both data manipulation and timing information of processes.

1.2. HIGH-LEVEL AND LOW-LEVEL HDLS

We can define low-level HDLs as, HDLs that are similar to Verilog, since Verilog is the most common HDL which uses low-level methodologies to model electronic systems. Low-level also means that it is close to metal than the high-level HDL. Just as in programming languages, where the assembly generated by a lower level language "C" is much more closer to the expressions written in it, than a higher level language "Haskell". While there are solutions to security verification of low-level languages, such as SecVerilog [8] which can verify Verilog. High-level languages require new approaches for their verification.

It is best to give an example of Chisel that demonstrates high-level functionalities which are not available to Verilog.

Algorithm 1.1. High-level Chisel code

```
abstract class GenericFilter[T <: Data](dataType: T) extends Module {
   val io = IO(new Bundle {
      val input = Input( Valid(dataType) )
      val output = Output( Valid(dataType) )
}

class PredicateFilter[T <: Data](dataType: T, fun: T => Bool)
extends  GenericFilter (dataType) {
   io.output.valid := io.input.valid && fun(io.in.bits)
   io.output.bits := io.input.bits
}

object MultiDigitFilter {
   def apply[T <: UInt](dataType: T) =
      Module(new PredicateFilter(dataType, (x: T) => x > 9.U))
}

object OddFilter {
   def apply[T <: Uint](dataType: T) =
      Module(new PredicateFilter(dataType, (x: T) => !(x(0).toBool)))
}

class MultiDigitOddFilter[T <: UInt](dataType: T)
extends GenericFilter (dataType) {
   val multiDigit        = MultiDigitFilter(dataType)
   val odd               = OddFilter(dataType)

   multiDigit.io.input := io.input
   odd.io.input        := multiDigit.io.output
   io.output           := odd.io.output
}
```

Algorithm 1.1. demonstrates some of the high-level concepts provided by Chisel HDL. *OddFilter* and *MultiDigitFilter* use the same parent design *PredicateFilter*. The PredicateFilter is a generic filter form that outputs either true or false based on the provided function and bits. *MultiDigitOddFilter* appends these designs to create a new functionality. Object-oriented concepts such as classes, generics, and abstracts are used to define modules. These features allowed defined parts to be reusable for different purposes. Being able to pass a function as an input is also used, which is a concept adopted from functional programming. These features are not available at low-level HDLs (such as Verilog).

1.3. MOTIVATION

Motivated by the lack of noninterference verification for high-level hardware description languages, this research proposes a convenient method to verify a high-level HDL with model checking while taking the high-level transformations applied to the design into account. High-level concepts made it easier to describe complex designs but it also invalidated earlier work on security verification of hardware, because of their newly introduced workflow structures. On top of that, the previous state of the art techniques expected the designer to know about the SMT solver language that is being used for the verification process, in order to make use some of its features.

Information flow policies are defined by lattice [9] structures. State of the art techniques expects designers to define these lattices in the SMT language. Expecting these definitions from designer have two downsides. Firstly, the designer must learn an external language in order to effectively use the verification tool. Secondly, since the verification toolchain does not have the lattice definitions at HDL level, it is not possible for state of the art tools to make certain optimizations that can reduce verification processes search domain.

1.4. SCOPE AND AIMS

A system is made of a set of variables (registers and wires) and their interaction in between. Every variable is associated with a security tag which is drawn from the security lattice of the system. The information flow policy is defined by system's security lattice. Based on a variable's security level and the information flow policy, it may or may not get interacted with certain variables. A system is considered as secure if the variables with high-security tags can not be observed by changing the configuration of low-security tag variables.

There are several methods for a system to get compromised. The following type of attacks are out of our scope;

- Side-channel attacks.
    i. Side channel attacks are based on information gained from physical implementation of a system. Electromagnetic and heat emissions can both be an example of such an information. They vary their magnitude based on the operation of the system which can be leveraged by the attacker.
- Physical attacks.
    i. Physical attacks can be in a broad range (directly tapping internal circuitry) and are also out of the scope of this thesis.
- Timing attacks.
    i. Timing attacks include any form of information leak gained through measuring the time it takes for an operation. These kind of attacks are also out of the scope of this thesis.

We can define the treat as a software level adversary and has access to all information at the lowest security level. Every system will have its own lattice structure that strictly defines allowed information flow. The lowest security level is the bottom of the system's lattice.

Aims of this thesis are;

- Algorithmically verify noninterference of high-level HDL.
    i. Static security tagging.
    ii. Dynamic security tagging.
- Handle intermediate variables created by high to low transformations, which are not available to the designer, thus are not associated with any security tags.
- By transforming security lattice to different representation forms between verification process, abstracting the designer from final verification statements. This reduces possible faults and is easier for the designer.
- A method that traces untagged variables and finds the most suitable security tag for it.
- Throughout analysis of conditional branching in the system. Branching can be inside or parallel to other branches.

2. METHODOLOGY

SecChisel focuses on hardware security verification of architectures at design time. The final goal is to prove that the design, in form of HDL code, is secure. "Secure" is in the sense of the design holds noninterference properties, which are properties about confidentiality and integrity.

The following diagram shows the overview of the SecChisel workflow.

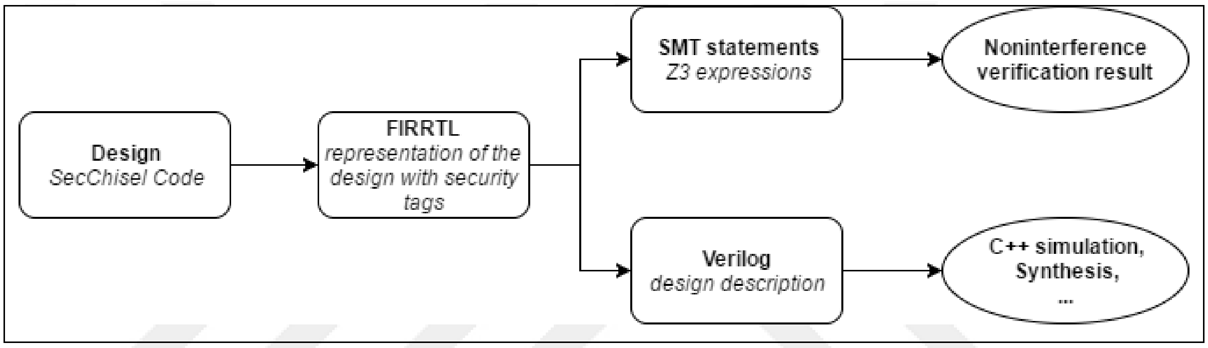

Design
SecChisel Code
FIRRTL
representation of the design with security tags
SMT statements
Z3 expressions
Noninterference verification result
Verilog
design description
C++ simulation, Synthesis, ...

Figure 2.1. SecChisel workflow

In the definition stage, the designer describes the system using SecChisel code (Scala). The designer is also responsible for providing an information flow policy over the system. This stage corresponds to the "**Design**" node in Figure 2.1.

Process stage includes parsing of FIRRTL output, handling generated intermediate variables and, generation of SMT expressions. It corresponds to "**FIRRTL**" node in Figure 2.1. The processing is done with Scala language and the expressions from FIRRTL language are the data that is being processed. The design is converted into a mathematical model at the end of this stage.

Lastly, at verification step, SMT solver tries to satisfy the generated mathematical model. SecChisel uses Z3 SMT solver. SMT-LIB defines the standard API for SMT solvers and Z3 follows these standards [10]. Z3 then either satisfies or concludes that the expressions are unsatisfiable which mean the design has either noninterference property or not.

In its core, SecChisel models the designs security information in SMT domain. All possible states the design can be in and their translations to different states are modeled in the SMT format. This new state representation of the design can be given to an SMT solver. While SecChisel acts like a code converter to a different code domain, it also takes into account of required theories the SMT solver will need.

3. BACKGROUND

3.1. FORMAL VERIFICATION

The need for formal verification stems from the ever-increasing complexity of both hardware and software systems. These systems become so complex that simulation and manual testing methods cannot cover extreme situations. Formal verification methods use mathematical techniques to generalize and reduce the problem space. Critical systems are getting successfully verified in order to avoid disasters. American space agency (NASA) successfully applied formal verification techniques to the control software of their deep space probes [11]. Microsoft also uses formal verification for their critical sub-systems. Formal verification is gaining ground as the ultimate testing technique for critical systems.

There are two opposing approaches to the verification problem. One of them is to examine all possible states the system can be in. Assessing each state with all possible combinations of external and internal stimuli. Checking that only the desired properties hold and the system can never in an invalid state. This can be categorized as an algorithmic approach to formal verification. The other option is the proof-based approach, which concentrates on using formalized mathematics. SecVerilog and Caisson [15] are examples of low-level languages that have verifiable information flow property that uses algorithmic approach.

SecVerilog operates on low-level Verilog code with a similar model checking approach of SecChisel. However, it does not verify the inner modules as SecChisel. SecVerilog assumes the inner module's port tags are provided by the designer and does not trace the ports through input to output. It is not possible to assign different lattices to inner modules with such approach. The tag propagation feature of SecChisel is also not available on SecVerilog. In the end, SecVerilog converts the information flow verification problem to SMT domain and verifies the design algorithmically.

Sapper HDL [16] verifies the information flow property at the language level. Sapper acts as a higher level language that generates Verilog code in the end. Information flow policies are enforced at the circuit level. While generating Verilog code, Sapper inserts checks that will detect any security violation. Any information flow violation would reflect as a functional bug. While Sapper acts as an abstraction above Verilog, it still does not provide high-level paradigms such as object-oriented and functional programming. Moreover, it does not generate any intermediate variables as in the case of SecChisel to FIRRTL. The provided abstraction is not in the sense of expressiveness, but in the sense of language level enforced security. The result Verilog design would have extra checks generated by Sapper, which results in overhead. The Sapper does not provide a formal verification in sense of SecChisel or SecVerilog. Compilation process generates tracking and checking logic based on formal semantics.

3.1.1. Theorem Proving

The system is modeled as a set of mathematical definitions as the first step of theorem proving. These become axioms of the system which then the desired properties of the systems are derived as theorems that follow these axioms. It is possible to classify theorem proves based on their underlying logic. A Certain class of theorem prover can be chosen based on the system under consideration. Classical theorem provers are based on classical higher order logic. This allows verification functions and functional programs to be relatively easier. HOL [12] and PSV [13] are well known classical theorem provers. Constructive theorem provers use constructive logics as their underlying strategy. The COQ [14] system is a constructive theorem prover.

3.1.2. Model Checking

Model checking is a state-based, algorithmic approach that proved its success. By modeling the state space, translations, desired, and undesired state configurations, the model checker proves if such undesired end results may ever happen. An assertion language is used to express desired properties. The mathematical assertion language is used to describe states of the system and their transitions. Algorithmic techniques for state explorations are then applied to search for whether a property holds true for states of the system. The model checker does not have to explore the entire state space in order to verify certain properties. Thus, it is possible to explore infinite state spaces with model checking techniques.

Model checking approach has several advantages. Once the correct representation of the system and the required properties are defined, the verification process is relatively fast and automatic. Secondly, if a property is not holding true for a state, the verifier is able to generate a counterexample, which is very useful in pointing the flaw of the system.

3.1.3. SMT

Satisfiability is the basic problem of determining if a set of constraint has a solution or a model. Different problems can be converted to their satisfiability equivalence, thus can be described in satisfiability problem domain. Graph problems, scheduling, software, and hardware verification are just a fraction of them. If a problem can be encoded by boolean formulas, then it can be solved by using a boolean satisfiability solver. Problems that require an extended expressiveness such as symbols, arithmetic, arrays, data types, uninterpreted function symbols can be handled by satisfiability modulo theories solvers. They are satisfiability solver with an extended expressiveness. SMT-LIB defines the common standards of SMT systems.

Algorithm 3.1. SMT verification example

```
(declare-fun x () Int)

(push)
(assert (< (* x x) 0))
(check-sat)
(pop)

(push)
(assert (>= (* x x) 0))
(check-sat)
(get-model)
(pop)

>> unsat
>> sat ➔ (model (define-fun x () Int 1))
```

Algorithm 3.1 proves "multiplying an integer with itself will always result in a positive number". Integer *x* is defined as any integer, without restrictions. Then model checker tried to satisfy the restriction in which multiplying *x* with itself has to be a negative number. SMT solver returned a *unsatisfiable* result for that assertion. This proved that there is no integer that by multiplying with itself can result in a negative number. The next assertion made SMT solver to find a number that results with a positive number after multiplying with itself. The solver was able to satisfy that assertion and returned the proposed model where the value of *x* is 1.

3.2. FLEXIBLE INTERNAL REPRESENTATION FOR RTL (FIRRTL)

FIRRTL is an IR for circuit designs. It is a platform for expressing circuit-level transformations. Chisel designs are converted to FIRRTL as the first step. While Chisel targets the FIRRTL language, other HDL languages can target FIRRTL. SecChisel encodes information flow in FIRRTL language and uses these encodings for verification. Another HDL can also target FIRRTL with SecChisel encodings in order to benefit from verification pipeline.

FIRRTL language can be parsed and processed as a statement/expression tree. This data structure is perfect for processing, manipulating, and extracting information from the design. As Chisel itself uses this data structure to convert designs to their lower level equivalences. A method called "Pass" is provided for passing over all of the design components and to do a certain task. The pass is an encapsulated map function that visits all statements and expressions of the circuit.

SecChisel processes FIRRTL language before generating SMT expressions. In order to follow specifications of FIRRTL language, SecChisel is not modifying the grammar of FIRRTL. SecChisel takes advantage of "info" token of FIRRTL language. The "info" is an optional token in most of the FIRRTL grammar rules.

```
Info      → FileInfo
FileInfo → '@[' ('\\]'|.)*? ']'
```

Figure 3.1. Info token lexer rules

Figure 3.1 shows the lexical rule of “Info” token. It is provided as a regular expression. Example Info tokens can be, “@[]”, “@[this is an info token]”, and “@[1234]”.

Info tokens are parsed along with their parent statement and made available for passes in the statement tree. SecChisel encodes and decodes every information about noninterference to these info tokens. They do not effect the end circuit in any way.

```
stmt     → 'wire' id ':' type info?
stmt     → 'reg' id ':' type exp ('with' ':' reset_block)? info?
stmt     → 'node' id '=' exp info?
circuit  → 'circuit' id ':' info? INDENT module* DEDENT
```

Figure 3.2. Example FIRRTL grammar rules that use optional Info token

4. DESIGN

This chapter discusses the design of SecChisel. It also defines underlying principles that it is built upon.

4.1. MODULE

Every design in Chisel is made out of modules. The circuit can contain one or multiple modules. Each module also defines their input and output ports which can be accessed by other modules. There are several ground data types that Chisel allows to be used in modules. These ground data types and their derivatives in modules are captured during the transformation to FIRRTL phase and converted to their appropriate expressions in FIRRTL language. Every ground type definition and their interactions with other ground types make the module's definition.

Algorithm 4.1. Basic input to output module in Chisel

```
class  InOut extends Module {

   val io = new Bundle {
      val x = UInt(INPUT, 32)
      val y = UInt(OUTPUT, 32)
   }

   io.y := io.x
}
```

Algorithm 4.1 shows a very simple circuit definition in Chisel. It consists of only an input variable *x* and an input variable *y*, where the output *y* is connected to the input *x*.

4.1.1. SecModule

SecModule is the base module class of every SecChisel design. The “SecModule” is an extension of “Module”. It gives several new abilities to Chisel such as, tagging a variable with security element and associating an information flow policy with the design.

The designer is responsible for using SecModule's new functionalities in order to give the clear definition of required information flow. Transformations of the modules to lower representations are abstracted from the designer.

Algorithm 4.2. Basic input to output SecModule in SecChisel

```
class  InOut extends SecModule {

   val io = new Bundle {
      val x = UInt(INPUT, 32)
      val y = UInt(OUTPUT, 32)
   }

   io.y := io.x
}
```

4.2. LATTICE AS INFORMATION FLOW POLICY

The designer is responsible for specifying an information flow policy by defining a lattice, consisting of security tags. Registers, wires and other components are associated with security elements drawn from the lattice. Security tags, their association, and relations are all designer-specified.

It is convenient to use lattice for information flow policy for several reasons. Using a lattice enforces there are a globally high and low-security elements. Moreover, any interaction between arbitrary variables can be resolved because of supremum and infimum guarantees made by lattice structure while not forcing the designer to specify relations between all possible combinations of security elements.

4.2.1. Lattice

Lattice is an algebraic structure with a set *S* and binary operations ∨ and ∧ defined on *S*. Let *S* be a set. Lattice *L* formally defines ordering between elements of *S*. Lattice is a partially ordered set with additional properties. These additional properties are a unique supremum and infimum result for every pair of elements drawn from the set *S*.

L is a lattice if the following identities hold for all elements *a, b, c* from *L*;

$a \vee a = a$	$a \wedge a = a$
$a \vee b = b \vee a$	$a \wedge b = b \wedge a$
$a \vee (a \vee b) = a$	$a \wedge (a \wedge b) = a$
$a \vee (b \vee c) = (a \vee b) \vee c$	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$

4.2.2. Bounded Lattice

Bounded lattice is a special lattice that has the greatest (1) and the least (0) element. For every element *x* in lattice *L*, $0 \leq x \leq 1$. Identity laws are introduced on join (∨) and meet (∧) operations by the bounded lattice.

The bounded lattice is an algebraic structure consisting of *L,* ∨*,* ∧*, 0*, and, *1*. Let *x* be an element drawn from the lattice.

$$x \vee 0 = x$$

$$x \wedge 1 = x$$

4.2.3. Supremum

Supremum is also referred as join, least upper bound or, ∨. Let *L* be a lattice and *S* be a subset of *L*. Supremum of subset *S* is the least element of *T* where *T* is the set of elements that are greater than or equal to all elements in *S*. A lattice *L* must have a single supremum result for all possible subset combinations where all subsets have exactly two elements.

4.2.4. Infimum

Infimum is also referred as meet, greatest lower bound or, ∧. Let *L* be a lattice and *S* be a subset of *L*. Infimum of subset *S* is the greatest element of *T* where *T* is the set of elements that are less than or equal to all elements in *S*. A lattice must have a single infimum result for all possible subset combinations where all subsets have exactly two elements.

4.2.5. Lattice Example

Let S be a partially ordered set. S = {a, b, c, d, e, f}, with binary relation "less than". We can assume the elements {a, b, c, d, e, f} are corresponding to security tags where *f* is the global high and a is the global low-security tags. Relation between elements are as follows, $a < b, a < c, b < d, c < d, b < e, e < f, d < f.$

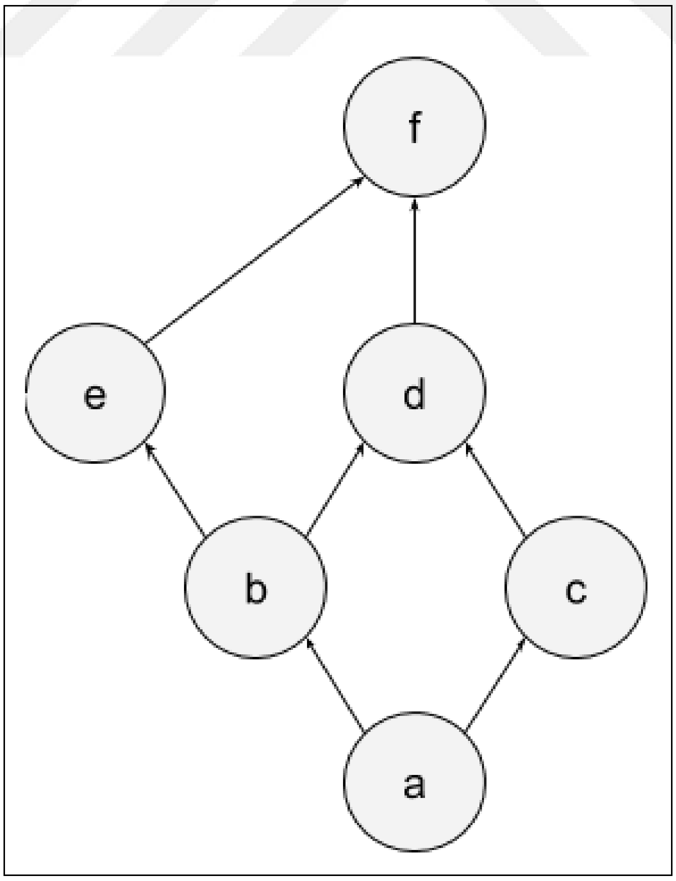


Figure 4.1. Example lattice structure

The figure above shows a graphical representation of the defined lattice structure. Every pair of element combinations have a single join and meet results and thus makes this partially ordered set a lattice. Since the lattice also have a global max (f) and global min (a), it is also a bounded lattice. Some of the supremum and infimum results are as follows;

$a \vee a = a$	$a \vee f = f$	$b \vee c = d$	$c \vee e = f$	$d \vee e = f$	$f \vee b = f$
$b \wedge b = b$	$e \wedge b = b$	$b \wedge c = a$	$e \wedge c = a$	$f \wedge c = c$	$e \wedge d = b$
$c \wedge e = a$	$a \vee b \vee c = d$	$c \vee b \vee e = f$	$f \wedge e \wedge d = b$	$b \wedge e \wedge d = b$	$a \vee b \vee c = a$

4.2.6. Comparing Different Lattices

It is possible that a design may contain more than one module that uses different lattices for their information flow policy. In such situations, the designer should map the external module's information policy lattice to his/her own. For each element in the inherited lattice, a map should be defined that relates those elements to their equivalent in the master lattice. A certain degree of heuristic can be applied to automate the process, such as the greatest and least elements of both lattices may be mapped directly to each other.

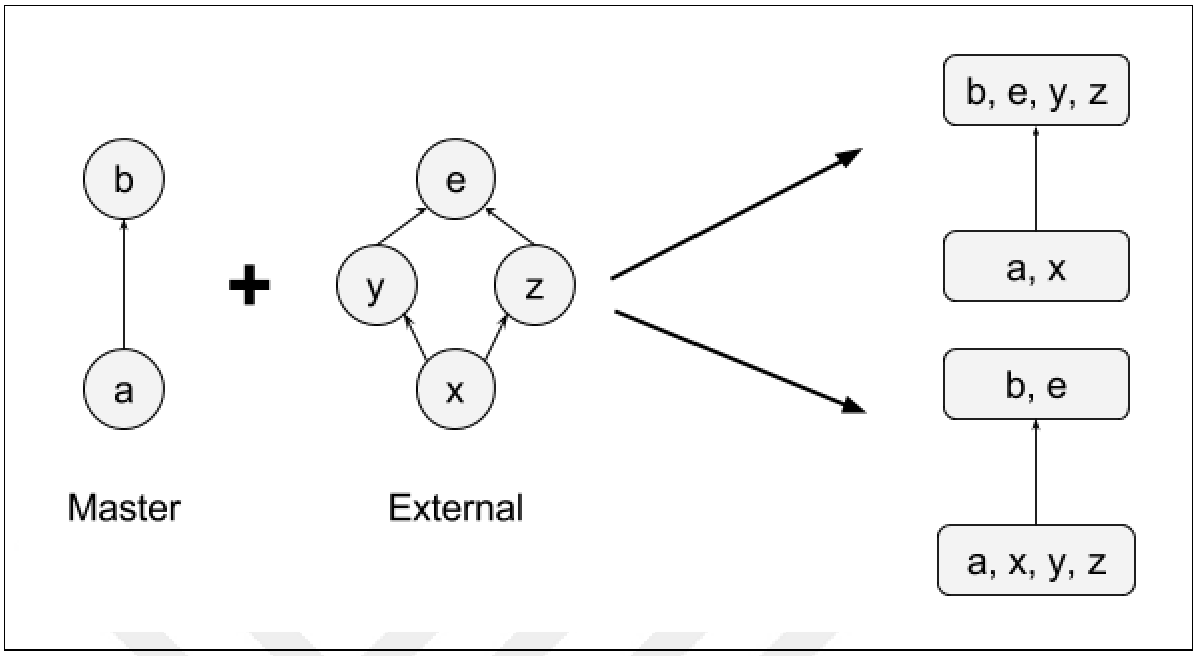


Figure 4.2. Ambiguous lattice map

However, as the figure above displays, there are possible configurations for master and external lattices that result in ambiguous maps. The designer must resolve this ambiguity by explicitly mapping external lattice to the master.

4.2.7. SecChisel Lattice

Every design in SecChisel have a lattice by default. It is possible to override and extend the default lattice. The default lattice consist of only "*High*" (global maximum) and "*Low*" (global minimum) security elements.

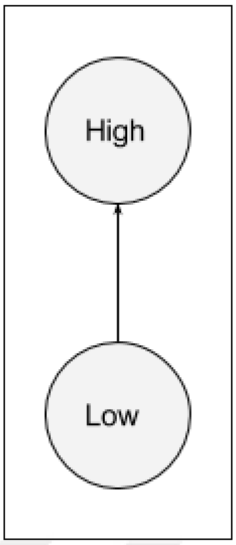


Figure 4.3. Default SecChisel lattice

In order to define a custom lattice, the designer must extend default lattice and introduce custom security elements and their relation with each other. All of the lattices are translated firstly to FIRRTL and then SMT statements respectively. The following figure demonstrate a custom lattice definition in SecChisel.

Algorithm 4.3. Diamond lattice

```
object DiamondLattice extends Lattice {

   val D1 = NewLatticeElement()
   val D2 = NewLatticeElement()

   LOW < D1 < HIGH
   LOW < D2 < HIGH

}
```

As Algorithm 4.3 shows, it is SecChisel provides an intuitive way to define new lattices. Lattice element relations are easily defined with “<” and “>” symbols.

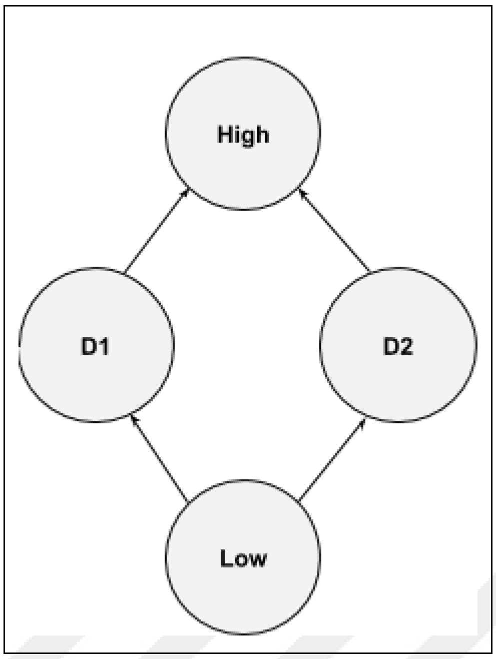


Figure 4.4. Custom diamond lattice

4.3. SECURITY TAGS

Lattice definition only defines the allowed information flow in the design. Variables of the design must also be appropriately tagged by the designer in order to fully describe the information flow of the system. SecChisel allows the designer to associate fixed security tags for each variable. The security tags are actually elements of the design's information flow policy lattice.

Designer can tag a variable statically or dynamically. Statically tagged variables will always have the same security tag throught the entire system whereas dynamically tagged variables may change their security tags depending on certain conditions. These two different kind of tagging mechanism allows flexibility to designer.

Chisel and its ground data types are modified in SecChisel in order to hold an association between variables and security tags. All tagging information is encoded to and decoded from FIRRTL language for processing stage of verification.

4.3.1. Static Tagging

The simplest tagging mechanism in SecChisel is the static tagging. It directly associates a given variable with a lattice element. The association will be valid in all conditions and can not be overriden. The syntax of static tagging is, "*variable := tag*". It is not mandatory to tag variables. All untagged are statically tagged with the least security element of their information policy lattice.

Algorithm 4.4. Static tagging example

```
class MakeAndOr extends SecModule {

  val IO = new Bundle {
      val x = UInt(INPUT, 1)
      val y = UInt(INPUT, 1)
      val andResult = UInt(OUTPUT, 1)
      val orResult = UInt(OUTPUT, 1)
  }

   // Static tagging
   IO.x             := LOW
   IO.y             := LOW
   IO.andResut   := HIGH

   IO.and          := IO.x & IO.y
   IO.or            := IO.x | IO.y
}
```

Algorithm 4.4 demonstrate static tagging of a SecModule. The "*IO*" bundle contains several *UInt* type variables inside it. *IO.x* and *IO.y* are statically tagged as *LOW* (the *LOW* element is drawn from the default lattice). *IO.andResult* is tagged with *HIGH* and while *IO.orResult* is not explicitly tagged by the designer it will be associated with static *LOW* tag. The ordering of static tag declarations are not important. It is not valid to statically tag a variable twice, as by definition static tags are consistent throughout the entire system.

4.3.2. Dynamic Tagging

Dynamic tagging allows variable to have a range of security tags at once. It allows designer to verify shared resources. For example, a dynamically tagged variable x can be both *LOW* and *HIGH*. These two tags may not always apply for the variable x, depending on the conditions. Possible tags may get eliminated from dynamically tagged variables in certain code branches.

Dynamic tagging requires a target variable, a dependent variable, and a tag range. The target variable is the variable that is getting tagged. The dependent variable and tag range are closely related. Tag range acts like a map function where it maps integer x to the lattice element l.

$$x \in \mathrm{N}, l \in L : \mathrm{f}(x) \rightarrow l \tag{4.1}$$

All function definitions that follows specification of formula 4.1 are valid map functions for dynamic tagging. Defined function needs to map natural numbers to lattice elements. These map functions are represented as interval maps rather than direct function definitions in the host language.

There are several reasons to use an interval map over pure function definitions. SecChisel translates verification related data through multiple layers of different language domains. This process requires encoding and decoding of the data. Interval maps are easier to encode and decode through these layers rather than direct function definitions. The other reason is that, directly using a function would increase computation intensity tremendously, map function would need to get computed for as every possible value of the dependent variable. For a 32 bit dependent variable, this would mean $2^{\wedge 32}$ function calls.

Dependent variable, along with map function determines the final tag of the target variable. A dynamically tagged variable *x*, must be verified for all possible values dependent *y* can get in current scope with respect to map function *f.* For *n* bit dependent variable *y*, the verification process checks all $y^{\wedge n}$ configurations (assuming the current scope does not prevent any configuration of *y*). Synax of dynamic tagging is, "*variable := (range, dependent*)".

4.3.2.1. Interval Map

Also referred as "Tag Range", the interval map is a key component of dynamic tagging. It is a representation of a function that takes a natural number and returns a lattice element.

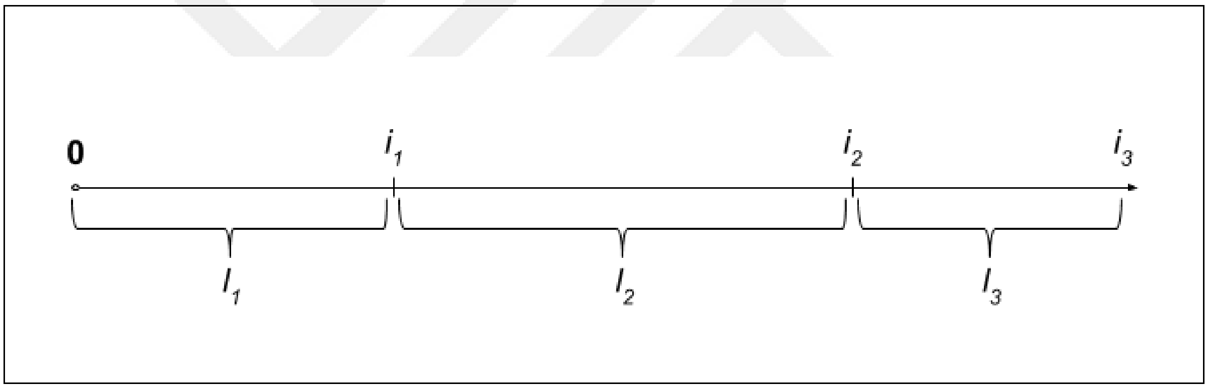


Figure 4.5. Interval map

Figure 4.5 shows a generic interval map representation. Interval ranges are denoted with 0, i_1, i_2, i_3, i_n and their corresponding lattice elements are denoted with l_1, l_2, l_3, l_n. Any value in range $[0, i_1]$ will be maped to li and $[i_{n-1}, i_n] \rightarrow l_n$.

As an example, supposing our *Diamond* lattice have the following security elements *low*, *high*, *d1*, and *d2*. It is possible to define the following interval map in SecChisel.

Algorithm 4.5. Interval range example

```
class TagExampleModule extends SecModule {

   val R1 =
      createTagRange(DiamondLattice.low).   // Out of bound tag
         add(0,    100, DiamondLattice.low).   // [0, 100] ➔ L
         add(101, 250, DiamondLattice.high).  // [101, 250] ➔ H
         add(251, 300, DiamondLattice.d1).    // [251, 300] ➔ D1
         add(301, 999, DiamondLattice.d2)     // [301, 999] ➔ D2
}
```

The Algorithm 4.5 results in following range maps, [0, 100] → *low*, [101, 250] → *high*, [251,300] → *d1*, [301, 909] → *d2*, and anything out of these ranges to *low*. Any query made to the R_1 would result in a lattice element *l* from *Diamond* lattice.

$R_1(10) = low$	$R_1(100) = low$	$R_1(150) = high$
$R_1(0) = low$	$R_1(333) = d2$	$R_1(500) = d2$
$R_1(260) = d1$	$R_1(999) = d2$	$R_1(9999) = low$

4.3.2.2. Dynamic Tagging Example

The dynamic tagging requires a tag range. The following example demonstrates dynamic tagging along with tag range definition.

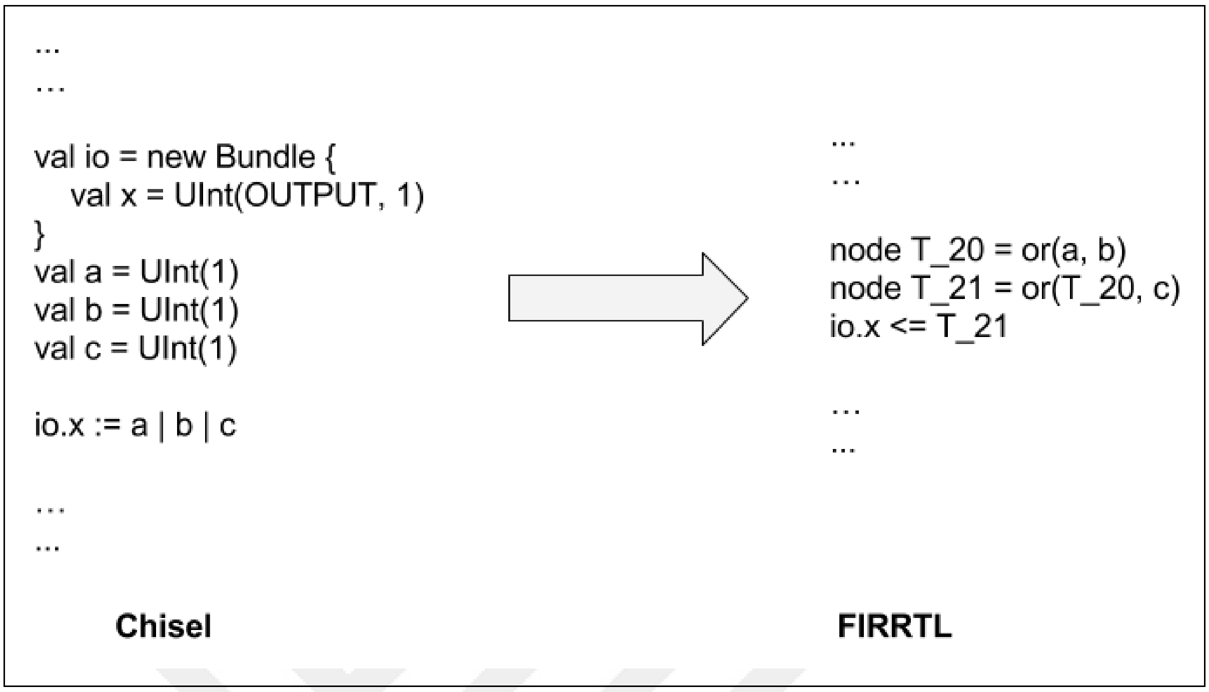
Algorithm 4.6. Dynamic tagging example

```
class DynamicTag extends SecModule {
  val IO = new Bundle {
     val y = UInt(INPUT, 1)
  }
  val d1 = UInt(2);
  val r1 = createTagRange(Low).add(0, 1, Low).add(2, 3, High)
  IO.y := (r1, d1);
}
```

IO.y is dynamically tagged with tag range *r1* and dependent variable *d1*. Security label of variable *IO.y* is not a single element but a set of security elements in this context. The bit width of the dependent variable determines possible inputs the tag range can receive. In this example that is, $2^2 = 4$ possible values. For values, 0 and 1 the tag range maps to the low security label and for values 2 and 3 the tag range maps to *high*. Since there are no restrictions to possible values *d1* can get, all of the 2^2 value range is considered. Resulting in *IO.y* to both *high* and *low* at the same time. Any verification will be done for both of these values.

4.4. CHISEL TO FIRRTL TRANSFORMATION

FIRRTL output is the middle ground between the design and the final output. Chisel allows complex expressions in a single line but FIRRTL does not allow such flexibility, thus the whole circuit goes through a series of transformation for their FIRRTL equivalence.



```
...
...

val io = new Bundle {
   val x = UInt(OUTPUT, 1)
}
val a = UInt(1)
val b = UInt(1)
val c = UInt(1)

io.x := a | b | c

...
...
```

```
...
...

node T_20 = or(a, b)
node T_21 = or(T_20, c)
io.x <= T_21

...
...
```

Figure 4.6. Chisel to FIRRTL example

Complex expressions get split and turned into binary operations with intermediate variables. Figure 4.6 demonstrate such transformation. The expression where the *io.x* variable get connected to the result of or operation between *a*, *b*, and *c* is expressed in three steps in FIRRTL. New nodes are also generated (*T_20* and *T_21*), which are not defined by the designer.

4.4.1. Encoding Scheme

SecChisel makes use of optional *Info* token of the FIRRTL language in order to encode its security information. Although the *Info* token allows a wide range of character usage in it, there is a case which results in a lexical error when parsing FIRRTL statements. The *Info* token starts with '*@[*' characters and ends with '*]*'. It is not allowed to use a '*]*' character inside *Info* token as the parser would catch this early closing bracket and interpret is as the end of *Info* token. This situation results in an error at parsing level. An example of such case can be “*@[1 2 3 [4 5] 6 7]*”. FIRRTL parser recognizes “*@[1 2 3 [4 5]*” as a full Info token and the rest “*6 7]*” violates FIRRTL grammar rules, thus a parser level error is raised.

SecChisel uses an encoding scheme that does not violate FIRRTL's grammar rules. The encoding is a recursive data structure which can be in three forms. These are the *object*, *list*, and *literal*. This encoding scheme will be referred as *Parsed Info* and its child data structures as *Parsed Object, Parsed List*, and *Parsed Literal* respectively.

Parsed literal is the simplest form a parsed info can be. It is just a plain string that does not contain *"}", "{", "(", ")"*, or *","* in it. Parsed literal can not reside any more parsed info structure inside of it. It is possible to describe them as leaves of the tree structure. Following strings are examples of parsed literals *"5<2", "static", "30", "low"*.

Parsed list is an enumerable data structure of parsed info. It is a list of parsed info where each element is separated by a "," character and are between parentheses. Following strings are examples of parsed lists *"()", "(1, 2, 3, 4)", "((a, b), (x, y), 5, 6, 7)"*. Since parsed list is also a parsed object, a parsed can contain another parsed lists inside it.

Parsed object is similar to parsed list in terms of structure. It is also an enumerable data structure with an addition. Each element inside the parsed object is associated with a key. The parsed object is a list of key-value pairs where the keys are strings and values are parsed objects. Series of key-value pairs are again separated with "," character and all of them are between brackets. Following strings are example of parsed objects *"{}", "{id:10}", "{numbers:(1,2,3)}", "{tag:high,ids:(1,2,3),type:{id:1}}"*.

SecChisel uses parsed info data structure to write and read from Info tokens of FIRRTL statements. Every security element such as lattices, dynamic tags, static tags and interval maps can convert their representation to parsed info structure. These security elements can also be reinitialized with an appropriate parsed info structure.

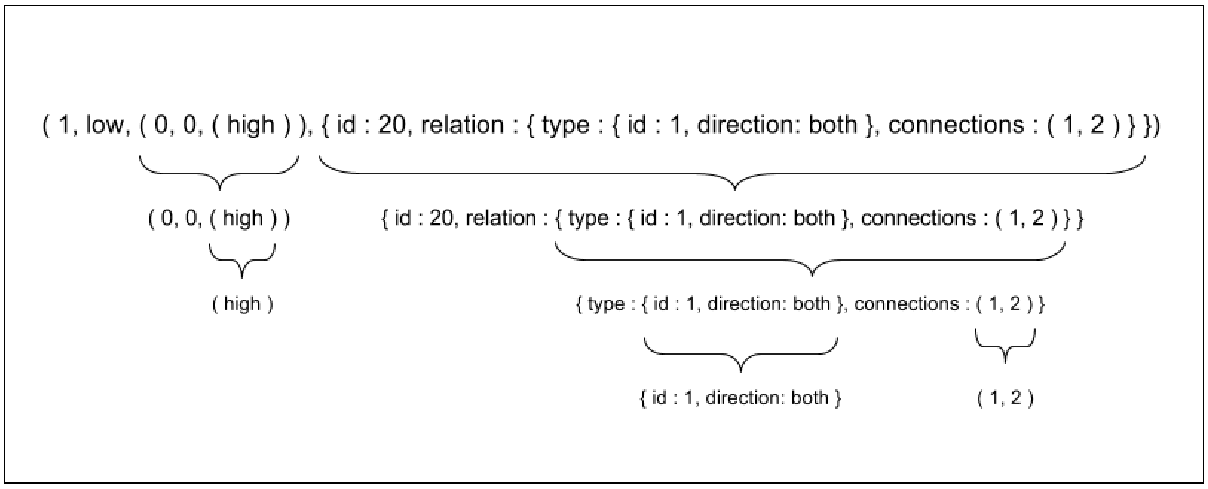

(1, low, (0, 0, (high)), { id : 20, relation : { type : { id : 1, direction: both }, connections : (1, 2) } })
(0, 0, (high))
{ id : 20, relation : { type : { id : 1, direction: both }, connections : (1, 2) } }
(high)
{ type : { id : 1, direction: both }, connections : (1, 2) }
{ id : 1, direction: both }
(1, 2)

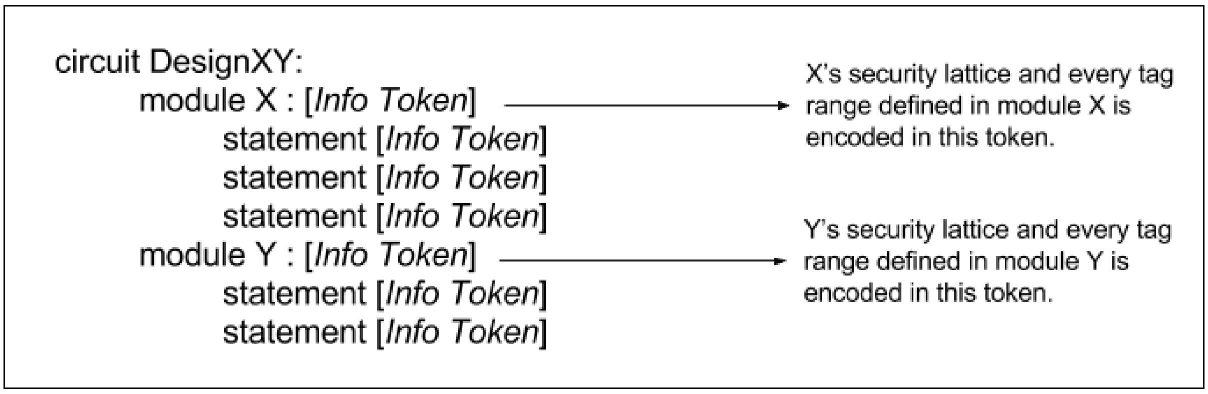
Figure 4.7. Parsed info breakdown example

4.4.2. Security Element Encodings

Lattices, dynamic tags, static tags and tag ranges are encoded with the parsed info encoding scheme. However, they are not encoded in a single info token. Location of encoding varies depending on which security element is getting encoded as these elements have a contextual meaning based on their location.

All SecModules have a security lattice. These lattice definitions are encoded in the “module” statement of FIRRTL. Each module begins with a “module” declaration which envelops every statement until the next module definition. Every module's security lattice is encoded in the info token of module statement and applies until the beginning of the next module definition. Each lattice definition has a unique identifier throughout the circuit for other security elements to refer to it.

Tag range definitions are also encoded in module definition's info token. These tag ranges are used by following statements as an instrument for dynamic tagging. They do apply until the beginning of the next module definition and have a unique identifier throughout the circuit.



```
circuit DesignXY:
    module X : [Info Token]
        statement [Info Token]
        statement [Info Token]
        statement [Info Token]
    module Y : [Info Token]
        statement [Info Token]
        statement [Info Token]
```

X's security lattice and every tag range defined in module X is encoded in this token.
Y's security lattice and every tag range defined in module Y is encoded in this token.

Figure 4.8. Lattice and tag range encoding locations in FIRRTL

Dynamic and static tag information are located in the info token of the variable declarations. Declarations can be in ports, nodes, and connects. In the processing stage of SecChisel, all of the tag information at these locations are fetched. A tag information contains the type of the tag (static or dynamic), identifier, and width of the related variable. Dynamic tags also encode dependent variables identifier along with tag ranges identifier used in the tagging process. Referred identifiers are linked with the security information gathered from the modules info token.

4.5. TRANSLATING PROPERTIES AND CREATING THE MODEL

Security properties are modeled in SMT language based on their FIRRTL representations. Lattice structures, lattice operations, static and dynamic tags are converted to SMT expressions.

4.5.1. Initial Definitions

SecChisel introduces a new data type to SMT named “tag”. The tag corresponds to security lattice elements. All of the lattice related functions are based on this newly introduced type. Lattice operations “less or equal” and “join” are also defined in the initial definitions.

$$x \in L, y \in L : \mathrm{LEQ}(x, y) \rightarrow \{0, 1\} \tag{4.2}$$

$$x \in L, y \in L, z \in L : \mathrm{Join}(x, y) \rightarrow z \tag{4.3}$$

$$\forall\, x \in L : \mathrm{LEQ}(x, x) = 1 \tag{4.4}$$

$$x \in L, y \in L, z \in L : \mathrm{LEQ}(x, y)\ \&\ \mathrm{LEQ}(y, z) \Rightarrow \mathrm{LEQ}(x, z) \tag{4.5}$$

$$\mathrm{x} \in \mathrm{L}, \mathrm{y} \in \mathrm{L} : \mathrm{LEQ}(\mathrm{x}, \mathrm{y})\ \&\ \mathrm{LEQ}(\mathrm{y}, \mathrm{x}) \Rightarrow \mathrm{x} = \mathrm{y} \tag{4.6}$$

Formulas above describe the initial properties of “less or equal” and “join” operations of lattice structures. Formula 4.2 and Formula 4.3 describes input and output types of *LEQ* and *Join* operations. Formula 4.4 states that for all elements in a lattice, applying the same element to the *LEQ* function returns *true*. Formula 4.5 implies that, if an element *x* is less than *y* and the element *y* is less than *z* results in *x* being less than *z*. Finally, Formula 4.6 says that if *x* is less or equal to than *y* and *y* is less or equal to than *x* , then *x* is equal to *y*.

Algorithm 4.7. Initial SMT statements

```
(declare-sort Tag)
(declare-fun LEQ (Tag Tag) Bool)
(declare-fun JOIN (Tag Tag) Tag)
(assert (forall ((x Tag)) (LEQ x x)))
(assert (forall ((x Tag) (y Tag) (z Tag))
   (implies (and (LEQ x y) (LEQ y z)) (LEQ x z))))
(assert (forall ((x Tag) (y Tag))
   (implies (and (LEQ x y) (LEQ y x)) (= x y))))
(assert (forall ((x Tag) (y Tag)) (= (JOIN x y) (JOIN y x))))
(assert (forall ((x Tag)) (= (JOIN x x) x)))
```

Algorithm 4.7 shows the translation of the mathematical rules that applies to lattices and operations defined over them, to SMT language. Programming in SMT language is different than conventional programming practices. Statements in the algorithm do not strictly define all behavior of these methods. The expressions only set certain restrictions over them in order to guide solver.

4.5.2. Lattice Modeling

Security information flow policy is represented as lattices. The lattice structures are also converted into SMT domain. Each security element is defined as a constant value and in order to limit the search space, result of join and LEQ operations are provided. Each lattice is a set, consisting of security elements. The lattice L is the following set, consisting of security elements s_i, $L = \{ s_0, s_1, s_3 \ldots s_i \}$. For every s_m, s_n combinations where $n \neq m$, elements s_n and s_m hold the following property, $s_n \neq s_m$. It is possible to let SMT come up with join operation results but modeling the lattice with direct join operation results have a positive impact on the computation intensity.

Algorithm 4.8. Initial SMT statements

```
declare-fun LE_2_HIGH () Tag)
(declare-fun LE_3_LOW () Tag)
(declare-fun LE_4_D1 () Tag)
(declare-fun LE_5_D2 () Tag)
(assert (not (= LE_2_HIGH LE_3_LOW)))
(assert (not (= LE_2_HIGH LE_4_D1)))
(assert (not (= LE_2_HIGH LE_5_D2)))
(assert (not (= LE_3_LOW LE_4_D1)))
(assert (not (= LE_3_LOW LE_5_D2)))
(assert (not (= LE_4_D1 LE_5_D2)))
(assert (= (join LE_2_HIGH LE_3_LOW) LE_2_HIGH))
(assert (= (join LE_2_HIGH LE_4_D1) LE_2_HIGH))
(assert (= (join LE_2_HIGH LE_5_D2) LE_2_HIGH))
(assert (= (join LE_3_LOW LE_4_D1) LE_4_D1))
(assert (= (join LE_3_LOW LE_5_D2) LE_5_D2))
(assert (= (join LE_4_D1 LE_5_D2) LE_2_HIGH))
(assert (lessorequal LE_5_D2 LE_2_HIGH))
(assert (lessorequal LE_4_D1 LE_2_HIGH))
(assert (lessorequal LE_3_LOW LE_2_HIGH))
(assert (lessorequal LE_3_LOW LE_4_D1))
(assert (lessorequal LE_3_LOW LE_5_D2))
```

4.5.3. Tag Range Modeling

Tag ranges required for dynamic tagging mechanism. They are represented as one-dimensional integer intervals to security element map at both Scala and FIRRTL levels. However, they are converted to a function definition at SMT stage. They are turned in to functions in following forms, $x \in \mathrm{N}, l \in L : \mathrm{f}(x) \rightarrow l$.

Let b_i represent the *i*'th bound of the tag range. Meaning, the area between b_0 and b_1 would represent the tag_1 which is the first map result of range.

$$\forall b_i \text{ where } i > 0, [b_{i-1}, b_i] \Rightarrow tag_i \tag{4.7}$$

Algorithm 4.9. SMT tag range definition example

```
(declare-fun TAG_RANGE_0 (Int) Tag)
(assert (forall ((x Int))
   (implies
      (and
         (>= x 0) (<= x 100))
         (= (TAG_RANGE_0 x) LE_1_LOW))))

(assert (forall ((x Int))
   (implies
      (and
         (>= x 101) (<= x 200))
         (= (TAG_RANGE_0 x) LE_0_HIGH))))

; Default Value for Range <0>
(assert (forall ((x Int))
   (implies
      (and
        (not (and (>= x 0) (<= x 100)))
        (not (and (>= x 101) (<= x 200))))
  (= (TAG_RANGE_0 x) LE_1_LOW))))
```

The example algorithm 4.7, by following formula 4.7 for each interval, models a tag range with two intervals, [0, 100] => *LOW*, [101, 200] => *HIGH* and everything outside these intervals to *LOW*.

4.6. NONINTERFERENCE VERIFICATION

Noninterference verification of the design is achieved by generated assertions for every data flow expression in the FIRRTL representation of the circuit. Successfully verifying all of these assertions means the design is verified as a whole. The data flow expressions of the FIRRTL language are node and connects. A node represents the result of a primary operation and there is an information flow from the parameters of the primary operation to the result node. The validity of information flow from the parameters of the primary operation to the node must be verified. Connect on the other hand is a direct information flow where there is no manipulation of data is applied. Each connect statements must be verified as well. There is also *conditions* which can generate additional verification statements if dynamic tagging is used.

Suppose there the variable v_0 is the left-hand side of an assignment operation and the v_1, v_2, v_3 … v_n are the right-hand side of the operation. The assignment operation means that there is an information flow from v_1, v_2, v_3 … v_n to v_0. This information flow must be check for its validity by asserting join of the right-hand side variables security tags is less or equal to the security tag of left-hand side variable. Si represents the security tag of the i'th variable. By this convention, the following formula shows the rule for verifying noninterference.

$$\text{Join}(s_1, s_2, s_3 \dots s_n) \leq s_0 \tag{4.8}$$

4.6.1. Untagged Variable Handling

SecChisel handles untagged and intermediate variables generated through FIRRTL transformation. The policy of deciding a tag for a variable depends on its context. Untagged variables created by the user gets the least security tag of their information flow policy lattice. Least tag *x* is an element which, for all elements in lattice *L*, $x < l$. Intermediate variables, on the other hand, inherits the security tag of their right-hand side. This rule is also aligned with formula 4.8 so that it does not require a further verification when such propagation happens.

$$s_0 = \text{Join}(s_1, s_2, s_3 \dots s_n) \tag{4.9}$$

Security tag propagation works in a similar way with dynamically tagged variables as well. Assume *R* is a tag range and *D* is a dependent variable. *RD* represents a tag range with a dependent variable. The security tag propagation can have a one or more *RD* in it. In this case, the join operation will not return a single security tag but a range of tags. For example, $s_0 = \text{Join}(R_1D_1, R_2D_2, R_3D_3, s_1)$, would result in s_0 to inherit the whole right-hand side with their tag ranges and dependent variables.

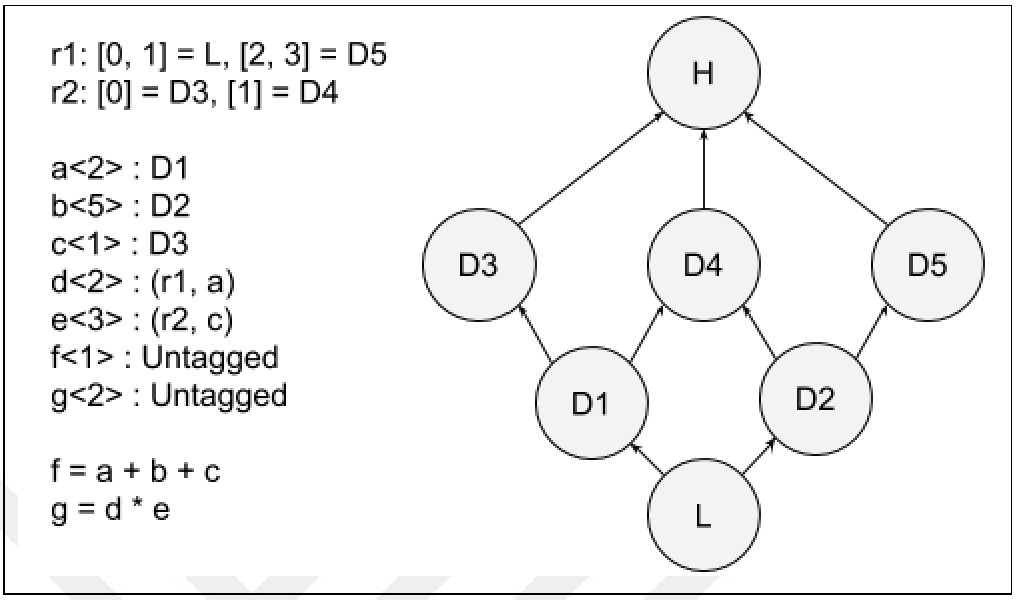


Figure 4.9. Security tag propagation example

Figure 4.9 shows a possible scenario of security tag propagation. Variables *a, b, c, d*, and *e* have their security tags assigned to them while *f* and *g* are untagged. Supposing that *f* and *g* are intermediate variables (not user-defined, but auto-generated nodes), they will have security tags propagated to them because of the following operations "*f* = *a* + *b* + *c*" and "*g* = *d* * *e*". The security tag of *f* will be *Join*(D1, D2, D3), which is *H*. Security tag of *g* will be *Join((r1, a), (r2, c))*, which results in a merged dynamic tag. The width of dependent variable *a* and *c* are also taken into account with their tag range and their possible security tags generate a new dynamic tag which is dependent to both *a* and *c*. Dynamic *tag (r1, a)* can result in *L* and *D5* while *(r2, c)* can result in *D3* and *D4*. The *Join* of combinations results in a new dynamic tag for *g* which is dependent on *a* and *c*. Any conditional branching that filters the value space of either *a* or *c* would result in the elimination of possible security tags on *g* as well.

4.6.2. Static Tag Verification

Statically tagged variables verified with a straightforward method that follows formula 4.8. The join of right-hand side is checked if it is less or equal to left-hand side security level.

However, the generated SMT expression is not directly asking if the information flow is valid. The question generated for the solver is if the negated information flow is satisfiable with respect to security lattice. If an assertion is satisfiable then the negation of it must be unsatisfiable. The straightforward expression would result in a single model that satisfies the assertions. In order to verify that this expression is correct in all cases, the negation of the expression must not be satisfiable in any case.

Assume there are variables *x* and *y* where *x* is statically tagged as *HIGH* and *y* is statically tagged as *LOW*. The expression *"x := y"* would result in the generation of the following SMT expression, *"(assert (not (LEQ LOW HIGH)))"*. The SMT expression basically asks the solver if it can find a case where the *LOW* tag is *NOT* less or equal to *HIGH*. Such state can not be modeled as it would contradict with axioms and thus the solver would give "unsatisfiable" output. The unsatisfiable output is the expected result if there is no information flow violation in the design as the solver could not find any state that can generate a violation. However, "*y := x*" would result in *"(assert (not (LEQ HIGH LOW)))*" assertion. This expression asks if the solver can find any state in which *HIGH* is *NOT* less or equal to *LOW*. SMT solver can find such a model since it does not violate initial axioms of the lattice structure.

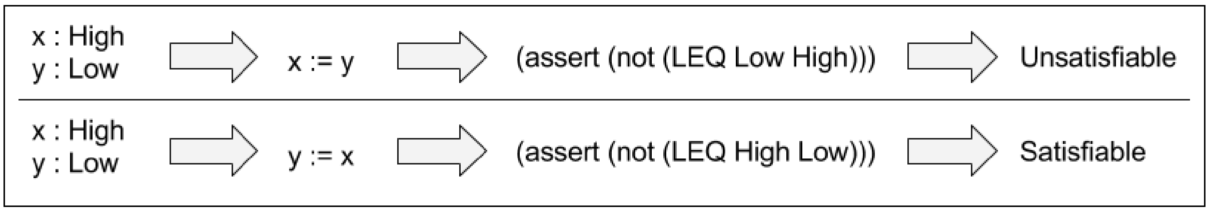


Figure 4.10. Static tag verification flow example

4.6.3. Dynamic Tag Verification

SMT assertions of dynamically tagged variables differ from static verification, but the process still follows formula 4.8. An information flow procedure may contain both statically and dynamically tagged variables at the same time. Dynamically tagged variables represent a set of security elements at the same time, which all must be verified for the same expression. In order to reason with dynamic tagging, a dependent variable with a map function must be present. The map function is in the following form, f(x) → *l*, where *l* is a security element and *x* is a positive integer. The dependent variables can be any variable in the design. Bit width of dependent variable is involved in the verification process as well as any restrictions on its value range on the current context.

Assume *x* is a dynamically tagged variable with dependent variable *y* and tag range *r*. In any assertion that requires to represents the tag of *x* will get the following expression;

$$D = [\ 0, 2^{\wedge W_y}\ ), \text{Tags} = \{\ r(d) \mid d \in D\ \} \tag{4.10}$$

Tags at Formula 4.10 represents the tag set of a dynamically tagged variable. The verification process is same with the statically tagged variable after substituting the static tag with "*Tags*" set in the verification expression.

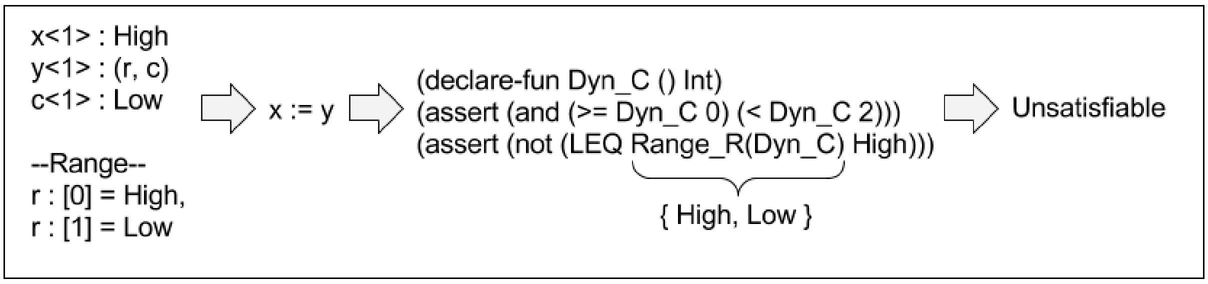

x<1> : High
y<1> : (r, c)
c<1> : Low
--Range--
r : [0] = High,
r : [1] = Low
x := y
(declare-fun Dyn_C () Int)
(assert (and (>= Dyn_C 0) (< Dyn_C 2)))
(assert (not (LEQ Range_R(Dyn_C) High)))
{ High, Low }
Unsatisfiable

Figure 4.11. Unsatisfiable dynamic tag verification flow example

Figure 4.11 demonstrate a right-hand side dynamically tagged variable in an assignment operation. Variable y is dynamically tagged with range r and it gets asserted for all possible outputs from *r* with input *c*. This results the dynamic tags to have both *High* and *Low* tags. This means the last assertion line is actually verified for both, *"(assert (not (Leq High High)))*" and *"(assert (not (Leq Low High)))"*. Which are both unsatisfiable, thus verified the noninterference in this line.

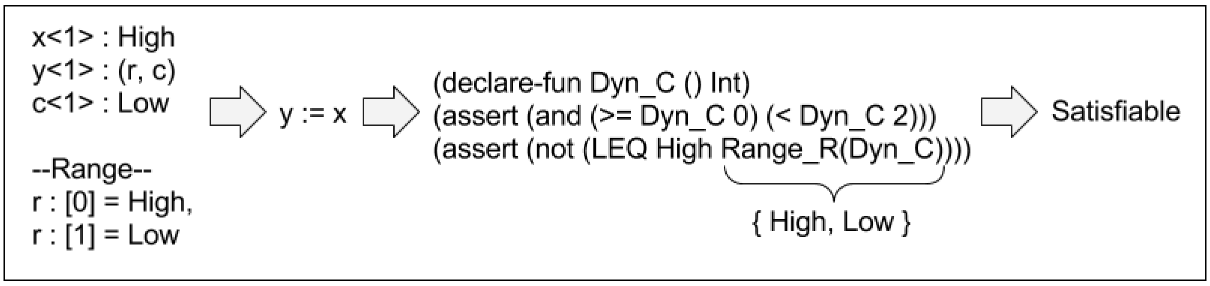

x<1> : High
y<1> : (r, c)
c<1> : Low
--Range--
r : [0] = High,
r : [1] = Low
y := x
(declare-fun Dyn_C () Int)
(assert (and (>= Dyn_C 0) (< Dyn_C 2)))
(assert (not (LEQ High Range_R(Dyn_C))))
{ High, Low }
Satisfiable

Figure 4.12. Satisfiable dynamic tag verification flow example

Figure 4.12 demonstrate a satisfiable case. It is enough to satisfy one case for dynamically tagged variables to conclude it satisfiable as a whole. While the *"(assert (not (Leq High High)))"* is unsatisfiable, *"(assert (not (Leq High Low)))"* is not, thus we conclude that the line "*y := x*" violates noninterference property.

4.6.4. Conditional Branching

Conditional branches have an effect on the verification process. The branching does not effect statically tagged variable verification but they are an essential part of dynamically tagged variables. Dynamically tagged variables are verified for every possible tag they can have which is determined by their tag range and dependent variable. Without any restrictions, all possible values of the dependent variable are taken into account for determining tag possibilities. However, if the target line that is getting verified is in a conditional branch which puts a restriction on possible values of the dependent variables value, then the possible tag set is reduced accordingly to the condition. Conditional branches may contain more conditional branches inside. Possibility set is reduced based on all of the conditions applies to the current context.

Assume the variable *x* is dynamically tagged. The dependent variable *y* has two bits, which means it is in the value range of [0, 3]. The tag range is [0, 2] = *Low* and [3, 3] = *High*. By default, *x* is both *High* and *Low*. Any verification must be made for both tags.

Algorithm 4.10. Conditional branching example

```
// Tag of x is High and Low
x := a

// Tag of x is High
if(y == 3) {
   x := a
}

// Tag of x is Low
if(y < 3) {
  x := a
}
```

Algorithm 4.10 demonstrate the default *x* tags which are both low and high at the same time. The condition applied to *y*, which is "*y == 3*", reduces the possible values *y* can be inside the condition. Variable *y* can only be "3" inside the first branch and since *y* is the control variable of *x*, this reduces the possible tags of *x* in that condition. The tag range returns *High* for value 3, which makes the tag of *x* as *High* inside the condition block. Next condition reduces the possible values *y* can have between [0, 3] and the tag range suggests that between values [0, 3] the tag can only be Low, thus the security tag of *x* is *Low* inside the second conditional block.

Algorithm 4.11. Mutiple conditional branching example

```
// Tag of x is High and Low
if(y > 0) {
   x := a
   // Tag of x is High
   if(y == 3) {
      x := a
   }
}
```

Algorithm 4.11 demonstrate multiple branching inside one another. The culling of dependent variables value space works the same. The first branch states that "*y > 0*", which only eliminates the value "0" from *y*. However, [1, 3] value space still generates both *High* and *Low* tags, thus the first conditional block did not cull any possible tags from *x*. The second conditional branch, however, forces *y* to be "3" which limits the value space of *y* to [3, 3]. This results in *x* having a *High* tag inside the second condition block.

4.7. INNER MODULES

SecChisel supports verification of inner modules. However, in order to verify the noninterference properly, inner modules must use the same lattice as information flow policy. Different lattices are incompatible for comparison, thus it is not possible to verify noninterference property of two modules that are using different lattices. By default, the inner module uses the lattice and security tags of its definition. An inner module may have a different context based on the used location of the design. For this reason, it is possible to lattice and tags of an inner module after its declaration.

A design definition may have multiple instances. All of the instances are translated into FIRRTL with their own unique tags and lattices. Each instance is verified individually. Having a replica of the base design as an instance allows SecChisel to distinguish each inner module's security configuration, as they are encoded in the FIRRTL. Normally, all modules have their variable tags in their FIRRTL definition. However, verification of inner modules adds a unique case to this generalization. Since inner module's IO port connections must be verified with its parent, the parent must access to variables and tags of the inner ones. After parent module’s port connections are verified with its corresponding instance of the inner module, the verification flow continues as usual. The instance of the inner module is then verified by its own as the verification process carries on.

5. IMPLEMENTATION AND EVALUATION

5.1. IMPLEMENTATION DETAILS

Implementation of SecChisel is done with Scala language. SMT language is also used at final verification process. Some Python scripts are also used for test design generation along with installation. In order to support security tag mechanism, some core files of Chisel has been changed. These files are Emitter, Data, and IR. Core data class of Chisel has been edited to contain security tag inside the object. The ":=" operator is also overloaded as tag assignment operation.

5.1.1. Tag Representation

As tags are cascaded through main variables to intermediate variables, a need to represent tags in a mergeable data structure arose. A tag in SMT domain can either be, static, dynamic or join or two different tags. Join operation may contain more join operation as its first or second parameter.

Algorithm 5.1. Tag data structure

```
abstract class Z3Tag(){}

class JoinZ3Tag(first: Z3Tag, second: Z3Tag) extends Z3Tag {
   override def toString() =
      s"(${Z3Constants.JOIN} ${first.toString} ${second.toString})"
}
class StaticZ3Tag(tag: LatticeElement) extends Z3Tag {
   override def toString() =
      Z3Constants.latticeElementToZ3(tag)
}
class DynamicZ3Tag(range: TagRange) extends Z3Tag {
   override def toString() =
      Z3Constants.tagRangeToZ3(range)
}
```

As data structure defined in Algorithm 5.1, the tag representations are held in either of these 3 forms. "*JoinZ3Tag*" receives two *Z3Tag* as input which enables this data structure to act as a tree data structure.

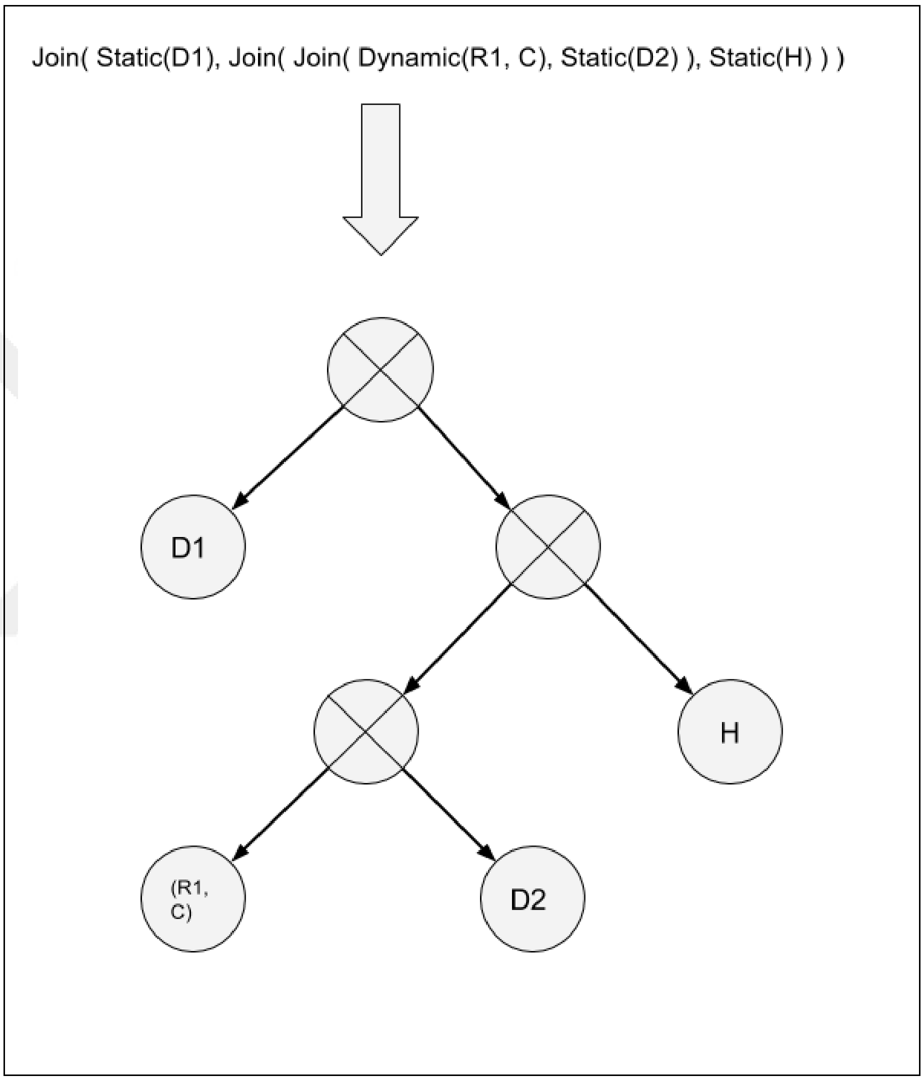

Join(Static(D1), Join(Join(Dynamic(R1, C), Static(D2)), Static(H)))
D1
H
(R1, C)
D2

Figure 5.1. Tag data structure example

Figure 5.1 demonstrate an example tag data structure with its graphical representation. Every join operation act as a node in a binary tree and its children might be a join, static, or dynamic tag. This data structure makes it easier to propagate tags to intermediate variables, as it is easy to append new nodes to a binary tree.

5.1.2. Tag Propagation

Tag propagation is applied on intermediate variables that are generated by Chisel while transforming the high-level description of the design to low-level. Tag propagation process begins when a *Node* definitions left-hand side is not tagged.

Algorithm 5.2. Right-hand side merge

```
def Merge(rhs: ListBuffer[String]): Z3Tag =
   if(rhs.length == 0)
      new StaticTag(lattice.min)
   else if(rhs.length == 1)
      fetchSecurityLevel(rhs.head)
   else
      new JoinTag(
         fetchSecurityLevel(rhs.head),
         Merge(rhs.tail)
        )
```

When an untagged *Node* is detected, right-hand side variables security levels are merged with algorithm 5.2. If the right-hand side does not have any variables then the algorithm picks the minimum lattice element. Min and max elements of lattices are determined with Algorithm 5.3.

Algorithm 5.3. Lattice min and max

```
def max(): LatticeElement =
   latticeElements.
   reduceLeft((le_1, le_2) =>
      if(le_1.isGreater(le_2)) le_2 else le_1
   )

def min(): LatticeElement =
   latticeElements.
   reduceLeft((le_1, le_2) =>
      if(le_1.isLess(le_2)) le_2 else le_1
   )
```

If the right-hand side has one variable, the *Merge* algorithm only fetches the security level of the variable end returns it. When there is more than one variable, *Merge* algorithm fetches the first variables security level and recursively calls itself with rest of variables. The first variables security level and the rest result are merged with *Join*.

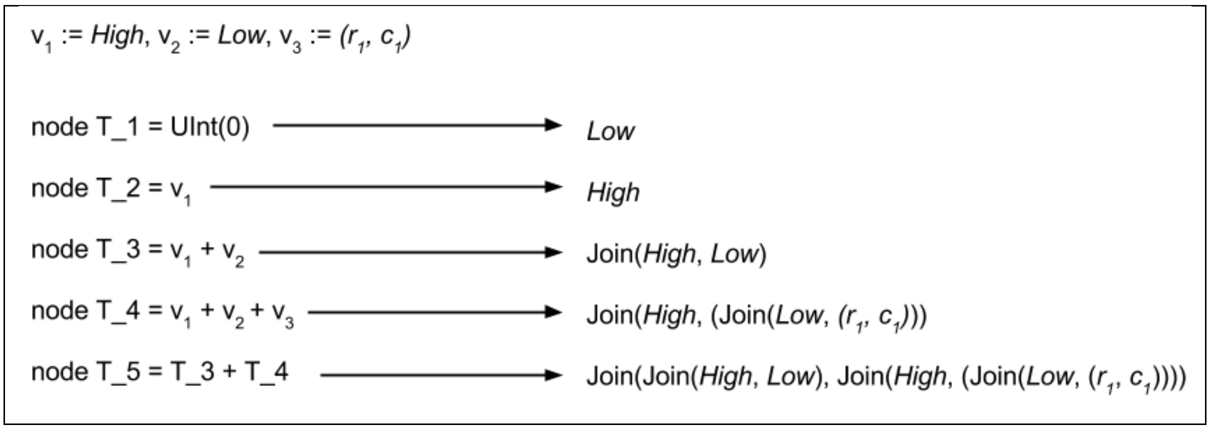

v_1 := *High*, v_2 := *Low*, v_3 := *(r_1, c_1)*
node T_1 = UInt(0)
Low
node T_2 = v_1
High
node T_3 = v_1 + v_2
Join(*High*, *Low*)
node T_4 = v_1 + v_2 + v_3
Join(*High*, (Join(*Low*, *(r_1, c_1)*))
node T_5 = T_3 + T_4
Join(Join(*High*, *Low*), Join(*High*, (Join(*Low*, (r_1, c_1))))

Figure 5.2. Propagation example

5.1.3. Join Lookup Table

In order to reduce the computation intensity of SMT solver, the results of join operations are provided to SMT domain. They can be deducted by solver with a cost on performance, but since our aim is not to prove the lattice structures properties it is beneficial to provide a join operation result table to SMT solver. Join results for all combinations of lattice elements are calculated for each lattice and translated to the SMT domain.

Algorithm 5.4. Greater elements of a lattice element

```
def greaterElements(): ListBuffer[LatticeElement] = {
   (directGreaterElements.clone() ++
     directGreaterElements.
       FoldRight(
           new ListBuffer[LatticeElement]())
           ((current, acc) => acc ++= current.greaterElements)
     ).distinct.sortWith((e1, e2) => e1 isGreater e2)
  }
```

Algorithm 5.5. Lattice join operation

```
def join(le_1: LatticeElement, le_2: LatticeElement):
LatticeElement = {
   val le_1_greater_or_equals = le_1.greaterElements()
   le_1_greater_or_equals.insert(0, le_1)

   val le_2_greater_or_equals = le_2.greaterElements()
   le_2_greater_or_equals.insert(0, le_2)

    return
      le_1_greater_or_equals.
         filter(e => le_2_greater_or_equals.contains(e)).head
}
```

Algorithm 5.6. Join lookup table generation

```
lattice.
  GetLatticeElements().
  combinations(2).
  map(pair => s"(assert (= (${Z3Constants.JOIN} " +
    s"${Z3Constants.latticeElementToZ3(pair(0))} " +
    s"${Z3Constants.latticeElementToZ3(pair(1))}) " +
    s"${Z3Constants.latticeElementToZ3(
      lattice.join(pair(0),  pair(1))
   )}))").mkString("\n")
```

Algorithm 5.4, 5.5, and 5.6 describes how SecChisel generates join lookup table for SMT domain. 2 combinations of lattice elements go through join operation and their results are saved for SMT solver. This ensures every possible join combination have a result in SMT domain. Join calculation is described in Algorithm 5.5. When two elements go through a join operation, their greater elements including themselves are taken into two separate lists. Then, the intersection of these two sets is taken and the first element is the joins result since the two sets are also sorted in ascending order.

5.2. PERFORMANCE EVALUATION

SecChisel aims to verify noninterference property of the designs at compile time without increasing compilation duration significantly. Generated SMT expressions are crafted with this mind side. The lattice structure is expressed as a discrete space and defining expressions avoid solver to come up with newly generated lattice elements. The verification process must be at least linearly dependent to the designs scale, in order to support large-scale designs.

Performance tests are made with generated circuits. It is possible to control the complexity of the design with a generator script. This creates a controlled environment where each attribute of the circuit can be observed for its impact on the verification process.

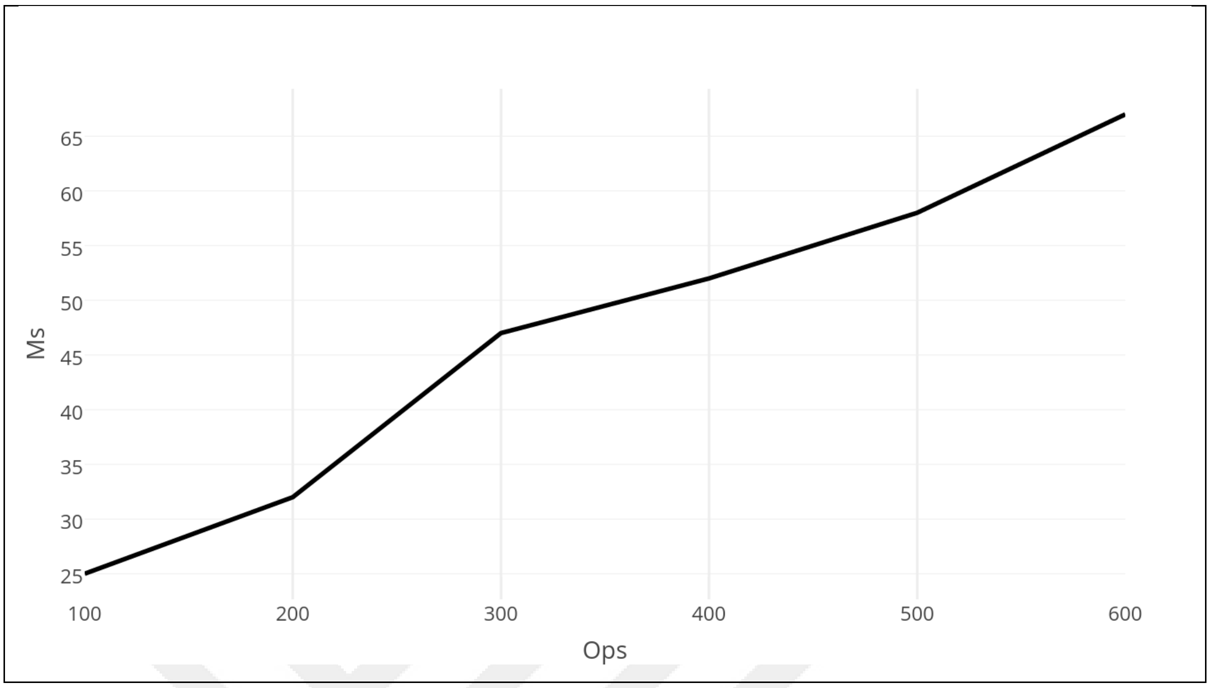


Figure 5.3. Static tag verification operations/milliseconds graph

Figure 5.3 shows benchmark results of verifying static tags. The *x*-axis represents the number of operations done in the design and the *y*-axis represents the milliseconds it took to verify it. Each test case is generated for 50 times and the interquartile average is taken as the sample. The number of operations is based on SecChisel and they are much more when they are converted to the FIRRTL language. As the figure shows, the time it takes to verify a fairly large design takes less than 100 milliseconds. It is also important to note that as the design scales, the time it takes to verify scales linearly dependent to it. The linear dependency shows that complexity of the design does not accumulate in the verification process. If it did, there would be a higher degree of polynomial dependancy. This makes sense because the SMT solvers stack is only filled with one expression at a time bundled with backbone rules. When the expression is asserted, it gets popped and the next one is added to the stack. For every static verification, the process boils down for the solver to determine if tag *x* is less or equal to than *y*, thus the complexity does scale linearly.

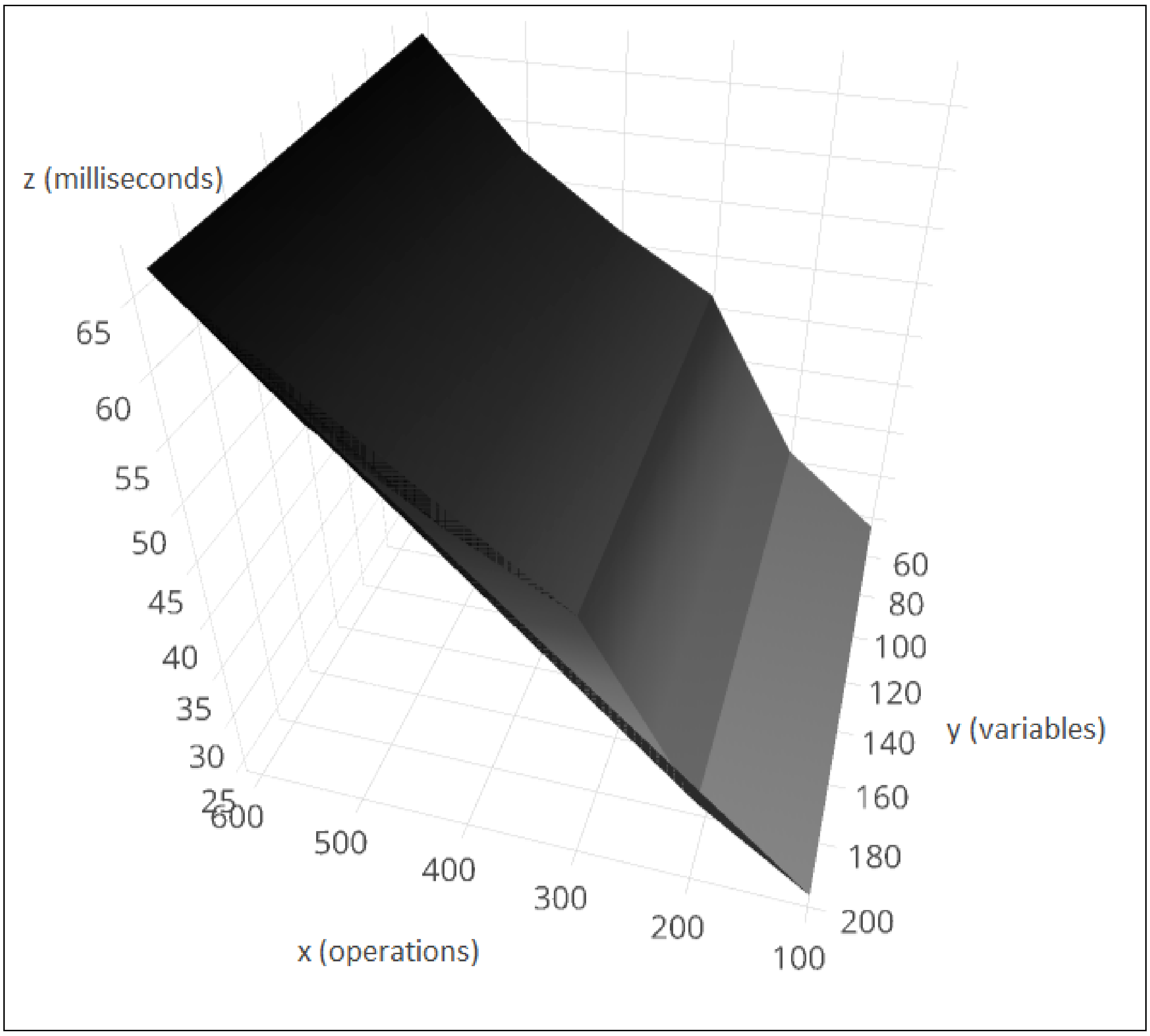


Figure 5.4. Static tag verification operations/milliseconds graph

Figure 5.4 add an extra dimension to the figure 5.3 by introducing a new parameter involved in the designs. The *x*-axis represents the number of operations in the design, the y-axis is the number of variables (register and wires) in the design and *z*-axis is milliseconds it took to verify the design for noninterference. For each “*x, y, z*” vertex position, 50 cases are sampled and their interquartile average is taken as position. It is observable in the figure that, the time (*z*-axis), purely depends on the number of operations in the design (*x*-axis). The additional dimension, number of variables, is insignificant in terms of verification time. As the number of variables increases, the time spent does not change for the same number of operations. This result is aligned with the design since when the design is transformed into the SMT domain, the variables are not represented with their names. They are not even translated to the SMT domain at all if they are not control variable for a dynamically tagged variable. Statically tagged variables are represented with their tag in the SMT domain and not with their unique names. As a result, increasing the number of statically tagged variables can not over accumulate the SMT domain, since they will be represented with their security elements. The information flow policy lattice tends to have a limited amount of security tags in it, even for complex designs. Even the most complex designs would not need an information flow policy with more than 20 elements. This means that even with the most complex design, all of the statically tagged variables in the design will be mapped to 20 possible values in the SMT domain at worst case.

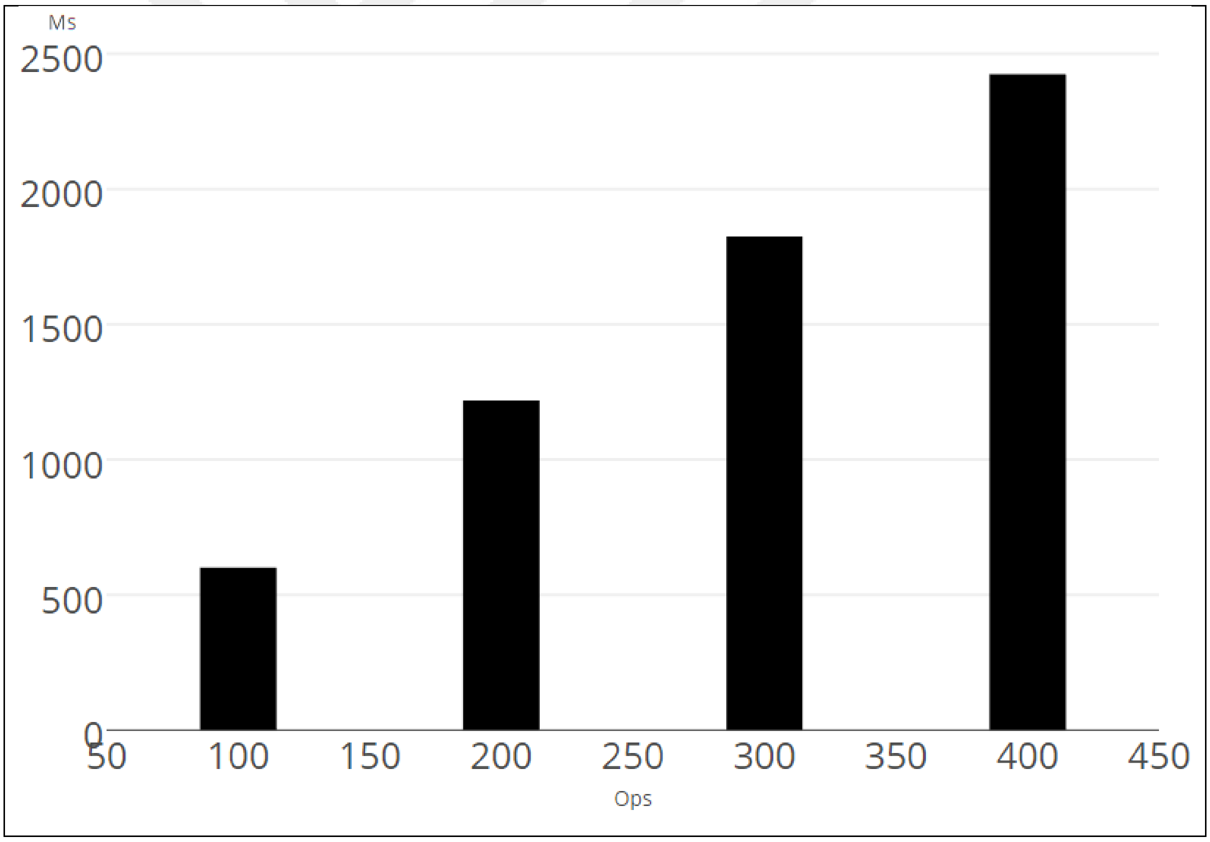


Figure 5.5. Dynamic tag verification operations/milliseconds graph

Figure 5.5 shows the dynamic tag verification performance chart. While verifying a dynamically tagged variable is significantly slower than the static counterpart, it is still an affordable duration as a compile time process. The linear dependency between operations and process time exists with dynamic tag verification as well. Dynamic tag assertions are also dependent on the provided interval range, control variable, and branch level. In order to test the effects of these attributes on the verification process, appropriate circuits are generated for testing purposes.

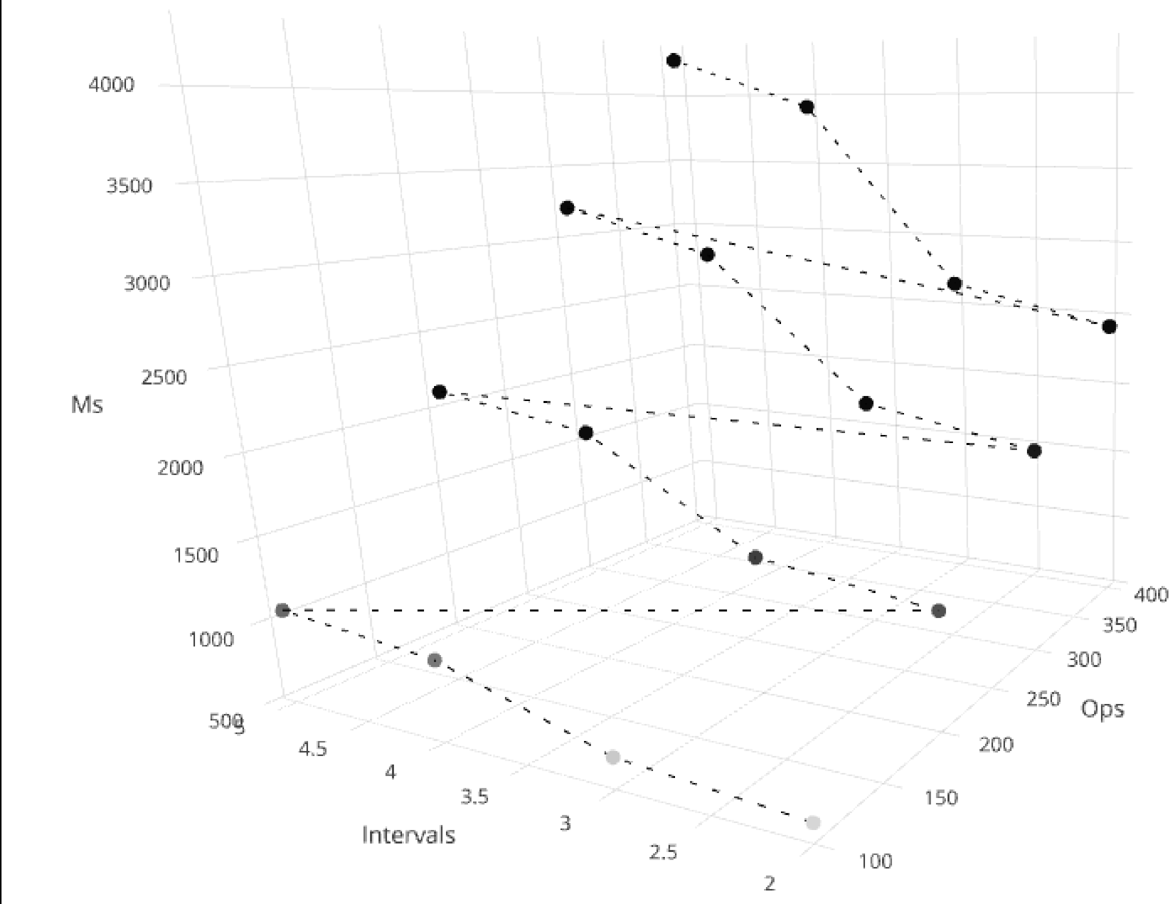


Figure 5.6. Dynamic tag verification with varying intervals

Figure 5.6 shows the impact of varying intervals on the performance of dynamic tag verification. Increasing the interval quantity have an observable impact on the verification process. The magnitude of the impact increases with the number of operations, but relatively the difference is 1.6 at maximum. Still, the most computation demanding circumstance only took 4 seconds to verify 400 dynamically tagged variables with 5 intervals.

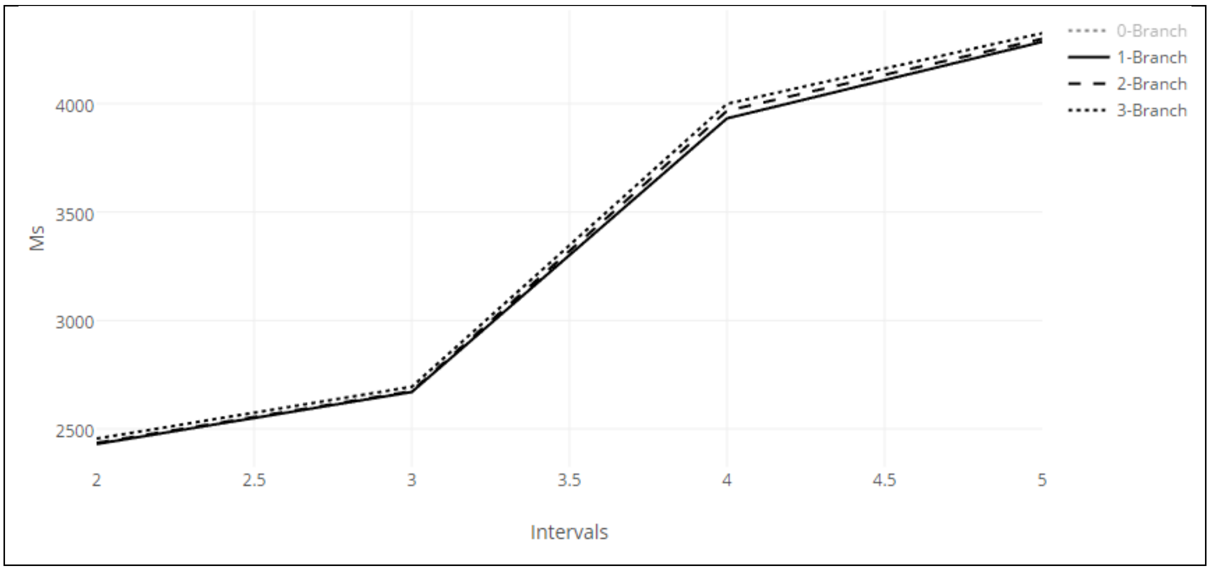


Figure 5.7. Dynamic tag verification with varying branches

Figure 5.7 shows the branching performance results of dynamic tag verification with varying intervals. All test cases for figure 5.7 are generated with 400 operations. Branching means a conditional case with a block of statements to be executed if the condition is true. 0 branch means the operations are not in a condition and 3 branch means the executed operations are inside 3 layers of condition cases.

Increasing intervals had the same effect with all branching types and the branching performance result can be ignored in all cases, as the difference between them is insignificant.

While test results show that SecChisel's verification process is affordable in terms of compilation overhead, it can still be improved. All of the performance tests are made on a single core. Information flow assertions are independent from one another and they can be parallelized. They only share the initial axioms provided at the beginning of SMT domain. The whole verification process can be divided into multiple SMT files with same axioms and can be verified on multiple cores, which in the end would hasten the process even further.

6. CONCLUSION

Design verification is one of the most important tasks at silicon development. As designs get more complex, informal verification methods, such as simulation-based techniques, has become insufficient to verify circuits. However, formal verification approach offers a generalized verification that can even detect edge cases in the design. Thus, formal verification becomes the default verification tool for critical system designs.

Hardware description languages are adopting high-level programming paradigms like functional and object-oriented programming. While high-level programming concepts make it easier to describe complex designs, it also creates a new problem with their verification. Most of the previous work on design verification assumed the hardware description language to be low-level. As a result of this assumption, new high-level HDLs can not directly benefit from previously proposed techniques for low-level HDLs. High-level HDLs transforms the design to lower level equivalences as it gets more concrete. During these transformations, intermediate components are automatically generated, which the designer have no control over.

We proposed SecChisel and techniques used in it to verify noninterference property of high-level HDLs. SecChisel formally verifies the design at compile time for its noninterference property. The design is transformed into SMT domain and proved for its noninterference property formally. Our work showed a convenient way to transform and the represent information flow policy in SMT domain, which is frequently used for security verifications. Static and dynamic tagging techniques enable designers to enforce strict and flexible rules for shared resources. Intermediate variables that are generated through high to low-level transformations are also handled by the propagation techniques we discussed. While previous works only assumed inner modules input and output ports security levels, our work handles inner modules and their noninterference by analyzing them entirely.

We have tested SecChisel for both correctness and performance. Benchmark results show that noninterference verification takes a fraction of a second at compile time and does not scale much, as the design gets more complex. Circuit design generator scripts are used for testing SecChisel. Various designs are used for correctness, especially for dynamic tag verification as the process is much more complex. The propagation of tags are also tested with auto generated designs with various complexities.

Our work showed a convenient way to formally verify noninterference property with an algorithmic approach. As a future work, timing channel analysis can be added to the SecChisel. Timing channel related information resides in the design through blocking and non-blocking operations and they can be transformed to the SMT domain. While SecChisel is based on model checking, a similar verification can target theorem proving based approach. Comparing the theorem proving to the model checking in terms of functionality and equivalency would be a beneficial addition.

REFERENCES

1. Intel Corporation, “4th Gen Core Family Desktop Specification Update”, http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf [retrieved 15 May 2017].

2. M. Hicks, C. Sturton, S. T. King, and J. M. Smith. Specs: A Lightweight Runtime Mechanism For Protecting Software From Security-Critical Processor Bugs. *ACM SIGPLAN Notices*, 50:517–529, 2015

3. Chisel Official Website. https://www.scala-lang.org/ [retrieved 15 May 2017].

4. Scala Official Website. https://www.scala-lang.org/ [retrieved 15 May 2017].

5. R. B. Lee. *Security Basics for Computer Architects (Synthesis Lectures on Computer Architecture)*, Morgan and Claypool Publishers, California, 2013.

6. D. F. Hsu and D. Marinucci. *a.6. Advances in Cyber Security: Technology, Operations, and Experiences*, Fordham University Press, New Yowk, 2013.

7. Verilog Official Website. http://www.verilog.com/ [retrieved 16 May 2017]

8. D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Turkey, 2015

9. D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 236-243, 1976.

10. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard Version 2.6, SMT LIB standard webpage. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-draft-3.pdf [retrieved 15 May 2017].

11. K. Havelund, M. Lowry, and J. Penix. Formal Analysis of a Space-Craft Controller Using SPIN. *IEEE Transactions on Software Engineering*, 27:1939-3520, 2001.

12. HOL-4 Proof Tool Official Website. http://hol.sf.net, [retrieved 22 May 2017].

13. S. Owre, J. M. Rushby and, N. Shankar. PSV: A Prototype Verification System. *Automated Deduction (CADE-11), 1992 Proceedings of the 11th International Conference on Automated*, London, 748-752, 1992.

14. Coq Official Website. https://coq.inria.fr/, [retrieved 22 May 2017].

15. X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardenkopf. Caisson: A Hardware Description Language For Secure Information Flow. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, California, 978-1-4503-0668-8, 2011.

16. X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A Language For Hardware-Level Security Policy Enforcement. *Proceedings of the 19th International Conference on Architectural Support For Programming Languages and Operating Systems*, New York, 978-1-4503-2305-5, 2014

17. O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of Approaches for Security Verification of Hardware/Software Systems, Cryptology ePrint Archive, Report 2016/846, https://eprint.iacr.org/2016/846.pdf, [retrieved 13 May 2017].

18. P. S. Li, A. M. Izraelevitz and, J. Bachrach. Specification for the FIRRTL Language. https://github.com/ucb-bar/firrtl/blob/master/spec/spec.pdf, [retrieved 30 May 2017]

19. J. Rushby. *Noninterference, Transitivity, and Channel-control Security Policies,* SRI International. Computer Science Laboratory, Ravenswood Avenue, 1992.