SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures

Shuwen Deng[1], Doğuhan Gümüşoğlu[2], Wenjie Xiong[1], Y. Serhan Gener[2], Onur Demir[2], and Jakub Szefer[1]

[1] Yale University, New Haven, CT 06510, USA
[2] Yeditepe Üniversitesi, 34755 Ataşehir/İstanbul, Turkey

ABSTRACT

Due to lack of practical and scalable security verification tools and methodologies, very few of the existing hardware-software security architectures have been thoroughly checked at the design time. To address this issue, our project develops a security verification methodology that is applicable to different hardware-software security architectures during the design phase. The verification framework aims to prove that a system holds desired properties with respect to not just functionality but also security; and we mainly focus on information flow and non-interference properties for verification. Using these properties, confidentiality and integrity of the sensitive data can be checked at design time. The proposed verification framework is built upon Chisel hardware construction language. By extending the Chisel language and tools, we created SecChisel. Ongoing work is focused on implementing SecChisel on top of Chisel 3 and realisation of the static and dynamic security labels.

1. INTRODUCTION

Top security researchers constantly remind us that, as computing becomes more pervasive, computer related security vulnerabilities are more likely to translate into real-world disasters [21]. In order to increase security of computing systems, many hardware/software security architectures have been designed. These architectures leverage special hardware features as trust anchors or provide security related functionality through new hardware features. However, if the designs are not perfect, they are still vulnerable to attacks, resulting in the failure of the promised protections.

Since the security of a computer system depends on the correctness of the protections that both the hardware and software components provide, there is the need to verify the security of both the software and the hardware components. Unlike software, hardware is almost impossible to patch once it is fabricated. Lack of sufficient formal verification at design time may leave some vulnerabilities in the manufactured system. For example, Intel's Core 2 Duo processor family is known to have 129 bugs [11] and many of the errata are security related [10].

The number of known bugs illustrates the limitations of the existing verification approaches. This has motivated researchers to look into the verification of the security properties of these systems at design time, and into developing new methods for the security verification of hardware and software systems.

Especially, formal methods provide the possibility of ruling out every security vulnerability in the designed system. Formal methods have been used in the *functional* verification of hardware and software for a long time. Recently, the use of formal methods for *security* verification of both the hardware and the software of a system has emerged as an important research topic.

This work aims to extend the field of hardware security verification by focusing on the hardware-software secure architectures that have been designed in academia [18, 22, 13, 14, 8, 4, 12, 23, 25] which provide enhanced security features in hardware; but most, if not all, of these designs do not come with any formal proof of security properties. The absence of formal verification of the security of system architectures may also be one reason why industry has been very slow to adopt the academic research in this field. A number of designs from processor vendors also provide some hardware features for security, e.g. ARM TrustZone [24], Intel SGX [19], and most recently AMD Memory Encryption [1]. Likewise, most, if not all, of these designs do not come with any formal proof of security properties.

1.1 Prior Work

A small number of projects, detailed in Section 8, have explored security verification of hardware architectures. These projects are shown in Figure 1, alongside the typical hardware levels in a computer system, and arrows showing at which levels the verification takes place. As SecChisel, all these projects focus on design-time verification. A large number of projects has been based

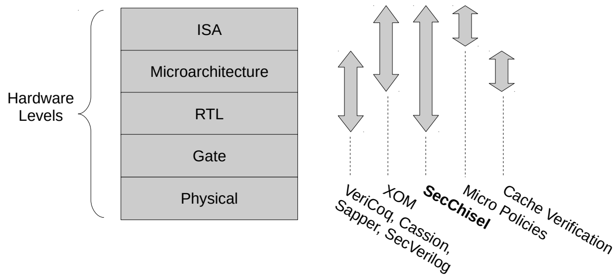

Figure 1: Overview of the typical hardware levels which a computer system can be logically broken down into, and the relation of the levels to existing security verification projects.

on the popular Verilog hardware description language. In addition, information flow tracking (IFT) or non-interference analysis are typical ways to reason about how information is processed by a system. Information flow tracking and non-interference can be used to enforce security properties such as confidentiality or integrity [20].
Goal of SecChisel is to further build upon, and extend, the existing work. This project is the first to explore use of Chisel [2] for security related work. Chisel provides number of high-level functionalities, including hardware design in a manner similar to object oriented programming which may make it more practical for new designers to get started with hardware design – and by extension with security verification when using SecChisel. The goal is for SecChisel to be a framework that can prove security correctness of designs with fine-grained, dynamic features, unlike prior work that focused on proving static cache partitioning or time-multiplexed pipelines.

1.2 Paper Organization

The remainder of the work is organized as follows. Section 2 gives overview of the SecChisel design. Section 3 discusses static security labels. Section 4 discusses dynamic security labels. Section 5 presents methodology for verification of designs with nested modules. Section 6 discusses possible optimizations, focusing on the SMT solving performance. Section 7 gives current status of the project. Section 8 presents the related work. Section 9 contains concluding remarks.

2. SECURITY VERIFICATION APPROACH OVERVIEW

Our SecChisel project focuses on hardware security verification of hardware-software architectures at design time. The goal is to show a design, in form of a hardware description language code, is correct. Issues of trusted CAD tools, hardware trojans, untrusted foundries, or supply-chain issues are not in the scope of this project. The hardware verification is done at design time, following the process depicted in Figure 2. The objective is to prove that a system representation holds desired properties with respect to not just functionality

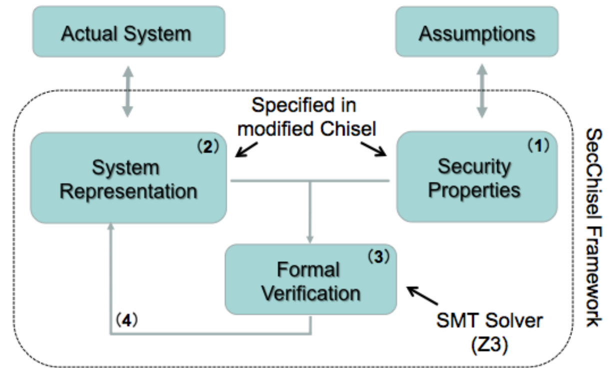

Figure 2: Security verification process considers security properties (1) and system representation (2), then verifies the design (3), and provides feedback to fix any issues (4).

but also security. Especially, our focus is on information flow and non-interference, which can be used to reason about confidentiality and integrity. As discussed next, SecChisel allows for system representation and security properties to be expressed in single code base – the modified Chisel code. This information is then used by an SMT solver, namely Z3, to check information flows based on the security properties. Current prototype focuses on explicit information flow, but implicit flows and timing will be explored in next steps of this research project.

2.1 Components of SecChisel

SecChisel is based on a modified **Chisel language and tools**. The base for the project is the existing Chisel 3, developed at Berkeley, and used for hardware construction. SecChisel extends data types with security labels, allowing designers to annotate design with the security labels associated with various wires and registers. The security labels come from a designer-specified lattice that can be used to reason about relationship of different security labels. The modified Chisel code is output (in form of modified FIRRTL) to the Z3 SMT solver, which checks for information flow violations based on the security labels.
Checking of the security properties is done using **Z3 SMT solver**. The focus is on using the algorithm approach that allows to ensure that information flow between components, wires, and registers does not violate expected security properties.

2.2 Lattice Model of Security Labels

To reason about information flow, the designer has to assign security tags to register, wires, or other components. The security tags come from a designer-specified lattice. Lattice is an algebraic structure that formally defines ordering between elements of a given set. It is a partially ordered set with additional properties. A partially ordered set is a lattice if it contains a unique supremum (join, also called least upper bound) and a unique infimum (meet, greatest lower bound) for every

pair of elements inside the set.
Supremum: Let P be a partially ordered set and S be a subset of P. Supremum of S is an element in P that is greater than or equal to all elements in S. It is also known as least upper bound. For a partially ordered set to be a lattice, the supremum operation must result in a single element for all possible subset combinations.
Infimum: Let P be a partially ordered set and S be a subset of P. Infimum of S is an element in P that is less than or equal to all elements in S. It is also known as greatest lower bound. For a partially ordered set to be a lattice, the infimum operation must result in a single element for all possible subset combinations.

2.2.1 *Bounded Lattice*

Bounded lattice is a lattice that has a greatest element (H) and a least element (L). For every element x in the lattice, L $\leq x \leq$ H. Bounded lattice introduces identity laws on join ($\wedge$) and meet ($\vee$) operations.

- Least element (L) becomes identity element for meet ($\vee$) operation. "X $\vee$ L = X"
- Greatest element (H) becomes identity element for join ($\wedge$) operation. "X $\wedge$ H = X"

2.2.2 *Lattice Example*

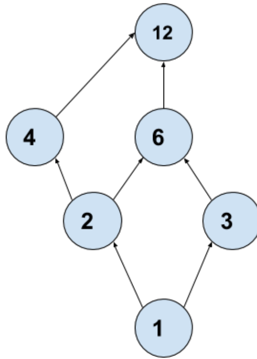

Figure 3: Example lattice structure

Let S be a partially ordered set. $S = \{1, 2, 3\ 4, 6, 12\}$, with binary relation, "divides". Here, the elements {1, 2, 3, 4, 6, 12} can correspond to the security tags. Now assume the ordering of elements with divides relation creates the structure shown in Figure 3.
And assume suprema are defined as follows: Supremum(1, 12) = (12), Supremum(1, 2, 3) = (6), Supremum(1, 3) = (3), Supremum(2, 4) = (4), Supremum(2, 3, 4) = (12). And assume infinea are defined as follows: Infimum(4, 6) = (2), Infimum(12, 4, 6) = (2), Infimum(4, 6, 3) = (1), Infimum(1, 12) = (1), Infimum(12, 4, 2) = (2). Given this structure, and given any two security tags, i.e. elements from {1, 2, 3, 4, 6, 12}, what is the join or meet of the two tags. Consequently, given inputs signals to a logic gate (or set of logic gates) each with different tags, the join or meet can be used to set the tag of output, depending whether considering confidentiality or integrity checks.

2.2.3 *Bounded Lattice as Information Flow Policy*

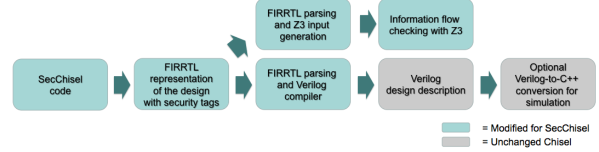

Figure 4: Overview of the SecChisel framework steps.

It is convenient to use a bounded lattice as information flow policy. It ensures there is a global confidential tag (H) and a global public tag (L) in the system. Any operation between two or more variables results in a value which has a security tag of the supremum of the operand's tags. Using lattice conveniently ensures that any possible combination of supremum will still result with a unique tag in the lattice set. In Figure 3 these correspond to elements 12 and 1.

2.2.4 *Comparing Lattices*

Modules in a design can come from different designers, who annotate their designs with security tags taken from different lattices. In such case the designer who integrates modules from other designers has to map one lattice onto another. Typically, designer may write a top master module, and integrate other nested modules within it. For each element in their master lattice, the designer has to specify relation to which element in the nested module's lattice it maps to. Some heuristics can be used to automate the processes. For example, nested module's lattice may only have two elements, high and low. These can be naturally mapped to the greatest and least element of the master lattice. In general case, however, it is an open question how to compare and translate between tags from different lattices. This work focuses on designs where all modules have security tags taken from same lattice; problem of different lattices will be explored in the future.

2.3 SecChisel Verification Steps

Given the lattice of security labels, designers can incorporate them into the hardware design through use of the SecChisel framework. The steps of the SecChisel framework are shown in Figure 4 and mirror the steps of Chisel, with addition of the Z3 STM solver. The goal of the framework is to show that none of the possible states of the design would violate the desired information flow policy. The policy itself is defined by the security labels from the lattice, and annotations that the designer puts in their design. Details of the specification of the security labels and how designer writes them down are given below, as well as in Section 3 for static security labels and Section 4 for dynamic security labels.
The annotated code specifies an information flow policy of allowed data transitions between wires, registers, or whole modules. A naive approach would be simulating every possible state of the design and try to catch an information flow violation. However, possi-

ble states of a circuit increases exponentially whereas it is variables increases linearly. A design made out of 100 variables (each of which are 1 bit) would have 1,267,650,600,228,229,401,496,703,205,376 possible states. Assuming we can verify each state in 1 nanosecond, verifying whole design in a brute force fashion would take more than 10^{13} years, thus this kind of approach is not feasible.

Meanwhile, our approach involves transforming the given design and information flow policy to its equivalent in SAT (Satisfiability Modulo Theories) domain. Once the problem is in a SAT domain, it can be solved by a SAT solver, such as Z3. The whole process can be divided in five steps:

1. Designer defines a design in SecChisel (Chisel with addition of new security labels).
2. SecChisel generates an intermediate level circuit defined in FIRRTL (Flexible Intermediate Representation for RTL). All information about the security labels is transferred in to FIRRTL.
3. FIRRTL output is parsed into a FIRRTL statement/expression tree. And, FIRRTL tree is processed into SMT statements understood by Z3 model checker.
4. SMT solver generates satisfiable or unsatisfiable result from SMT statements.

2.3.1 SecChisel Circuit Definition and Flow Policy

A design is defined in an identical way as with Chisel. As an addition to Chisel, the designer can also define a information flow policy for the circuit. A flow policy consists of security labels and a lattice. Security labels are just identifiers (string) that can be bound to any variable (wire or register) in a design and lattice defines the hierarchical structure between security labels. The security labels are taken from a lattice. The lattice must be a bounded lattice, which is a special type of lattice that contains a global highest and a global lowest elements. For confidentiality, it is not allowed for variables bounded with higher security labels to transfer their data in variables with lower security labels. For integrity, it is not allowed for variables bounded with lower security labels to transfer their data in variables with higher security labels. Each variable (wire or register) is then potentially labeled with two security labels, read security label, t_{read}, and write security label, t_{write}, both taken from the lattice. The designer adds these security labels into the source code, by annotating module inputs, outputs and individual wires or registers inside a module. Chisel's data types are modified in a way that user can give an extra parameter to their declarations to bound a security label to it. Not all wires or registers need to be labeled in a design, and the tools aim to infer some security labels. Underspecification can lead to false positives, and designers may have to update their design if the security checks fail due to wire or register with initially undefined security labels. The security labels from labeled portions will be propagated during synthesis and checking time, until a conflict is found, or no conflict exists.

The security label checking is effectively done at each assignment operation specified in Chisel code, although number of optimisations are proposed in Section 6. The designer can also explicitly declassify a signal, i.e. overwrite the default security label propagation rules. E.g. encrypted output of an encryption module can be declassified to indicate that the data has low security label (assuming correct implementation of the encryption algorithm). Special annotation needs to be added to the assignment during which declassification needs to be done. Opposite is also possible, security label's value can be increased, e.g. to allow an initially low valued signal to write to high valued register (otherwise integrity violation may be detected).

2.3.2 SecChisel to FIRRTL Transformation

FIRRTL language was created specifically for definition of digital designs. Chisel's output is an equivalent design definition in FIRRTL. FIRRTL language can be parsed into an expression/statement tree. Design in this data structure form is easier to be processed. SecChisel does not modify FIRRTL language's grammar to bind the security labels to their respective variables. FIRRTL statements, by their grammar definition, have an optional field for extra information that is ignored by circuit generation process. However this extra information fields are captured by parser and stored in their respective vertices in the tree structure. SecChisel uses this feature to transfer lattice structure and security label values in to FIRRTL definition.

2.3.3 Parsing FIRRTL Code

FIRRTL parser converts a stream of strings that grammatically satisfies FIRRTL language's rules, into a FIRRTL statement tree. The statement tree can be traversed and manipulated. Parsing of SecChisel's FIRRTL output is no different than Chisel's FIRRTL output.

2.3.4 Processing FIRRTL Tree

Processing the FIRRTL tree is the part where the problem domain conversion is made. First set of SMT statements defines lattice structure as a directional graph (directions are from lowest to highest). Thus making a path available among security labels that can be compared (a higher security label to a lower security label), but preventing any paths between uncomparable security labels. Then, during parsing, each node of the statement tree gets visited and an appropriate SMT statement is generated. The generated SMT statements can be divided in two parts.

First part involves the definitions and restrictions which the solver will have to satisfy for each of the statements on the second part. Second part involves statements for solver to satisfy. Up to two assertions are created when a statement in FIRRTL tree with data transition is encountered. For read security labels, t_{read}, the target variable's bounded security label must be greater or equal than the source variable's bounded security label for valid information flow (source's security label must

be less or equal than target's). If there is an operation between source variables (a = b + c), then source variable's bounded security label's least upper bound (join) must be less than or equal to target's security label. When such statements are encountered, the processor generates a path availability question for SMT solver to satisfy. If there is a violation in information flow, SMT solver will not be able to solve generated path requirement. The opposite checks are done for write security labels, t_{write}.

2.3.5 *Verification Using Z3*

Generated SMT statements are input to a SMT solver as final step. If SMT solver can satisfy all of the assertions, then all information flow operations satisfy initial flow policy. Unsatisfiable assertions means there is a violation of flow policy (no path exists in the defined graph) in that particular case.

3. STATIC SECURITY LABELS

SecChisel allows the designer to enter fixed security labels for each variable. Chisel's data types are modified in a way that user can give an extra parameter to their declarations to bound a security label to it. The security labels are "static" as each has explicit value assigned to it by the designer. There are also "dynamic" security labels, where the security label's value depends on a value of another signal. The dynamic security labels can help facilitate security checks when design has modules that are multiplexed between different entities. Dynamic security labels are explained in following Section 4.
There are several common data types in Chisel needed to be extended with ability to add security labels. Security labels added to the data types will be propagated from Chisel to the intermediate representation FIRRTL. The following are the ones needed:

- Basic datatypes: UInt, SInt, Clock
- Aggregate datatypes: Bundle, Vector
- Composite (original) datatype: Reg, Wire
- Other datatype: Bool, Reset, etc.

3.1 Static Security Labels in SecChisel Code

```
class FullAdder extends Module {
   val io = new Bundle {
      val a = UInt (INPUT, 1, NONE , ”0”)
      val b = UInt (INPUT, 1, NONE, “2”)
      val cin = UInt (INPUT, 1, NONE, “0”)
      val sum = UInt (OUTPUT, 1, NONE, “1”)
      val cout = UInt (OUTPUT, 1, NONE, “0”)
     }
     //Generate the sum
     val a_xor_b = io.a ^ io.b
     io.sum := a_xor_b ^ io.cin

     //Generate the carry
     val a_and_b = io.a & io.b
     val b_and_cin = io.b & io.cin
     val a_and_cin = io.a & io.cin
     io.cout :=
     a_and_b | b_and_cin | a_and_cin
}
```

```
circuit FullAdder :
   module FullAdder :
      input clk : Clock
      input reset : UInt<1>
      output io : {a : UInt<1>, b : UInt<1>, cin : UInt<1>, sum : UInt<1>, cout :
UInt<1>} @[SECTAGS:{cout:0,sum:1,cin:0,b:2,a:0}]

      io is invalid
      node a_xor_b = xor(io.a, io.b) @[FullAdder.scala 15:22]
      node T_5 = xor(a_xor_b, io.cin) @[FullAdder.scala 16:21]
      io.sum <= T_5 @[FullAdder.scala 16:10]

      node a_and_b = and(io.a, io.b) @[FullAdder.scala 19:22]
      node b_and_cin = and(io.b, io.cin) @[FullAdder.scala 20:24]
      node a_and_cin = and(io.a, io.cin) @[FullAdder.scala 21:24]
      node T_6 = or(a_and_b, b_and_cin) @[FullAdder.scala 22:22]
     node T_7 = or(T_6, a_and_cin) @[FullAdder.scala 22:34]
     io.cout <= T_7 @[FullAdder.scala 22:11]
```

Figure 5: Example of FullAdder with added security labels.

As a very simple example, we show implementation of a full adder. After adding security labels to the input and output port variables which have the types of UInt and Bundle, the code looks as in Figure 5. For each, there is no write security label, "NONE", and read security labels are specified as numerical values.
Given that now there are additional security labels in the input variables in Chisel, and they are propagated during the different assignment, and are added to FIRRTL representation of the circuit as well, as show in Figure 6. Notice Chisel adds some intermediate variables not originally in the design code, these require security label propagation as well.

```
class FullAdder extends Module {
   val io = new Bundle {
      val a = UInt (INPUT, 1, NONE , ”0”)
      val b = UInt (INPUT, 1, NONE, “2”)
      val cin = UInt (INPUT, 1, NONE, “0”)
      val sum = UInt (OUTPUT, 1, NONE, “1”)
      val cout = UInt (OUTPUT, 1, NONE, “0”)
     }
     //Generate the sum
     val a_xor_b = io.a ^ io.b
     io.sum := a_xor_b ^ io.cin

     //Generate the carry
     val a_and_b = io.a & io.b
     val b_and_cin = io.b & io.cin
     val a_and_cin = io.a & io.cin
     io.cout :=
     a_and_b | b_and_cin | a_and_cin
}
```

```
circuit FullAdder :
   module FullAdder :
      input clk : Clock
      input reset : UInt<1>
      output io : {a : UInt<1>, b : UInt<1>, cin : UInt<1>, sum : UInt<1>, cout :
UInt<1>} @[SECTAGS:{cout:0,sum:1,cin:0,b:2,a:0}]

      io is invalid
      node a_xor_b = xor(io.a, io.b) @[FullAdder.scala 24:23,
SECTAGS:{a_xor_b:2})]
      node T_10 = xor(a_xor_b, io.cin) @[FullAdder.scala 26:22,
SECTAGS:{T_10:2})]
      io.sum <= T_10 @[FullAdder.scala 26:10]
      node a_and_b = and(io.a, io.b) @[FullAdder.scala 30:23,
SECTAGS:{a_and_b:2})]
      node b_and_cin = and(io.b, io.cin) @[FullAdder.scala 31:25,
SECTAGS:{b_and_cin:2})]
      node a_and_cin = and(io.a, io.cin) @[FullAdder.scala 32:25,
SECTAGS:{a_and_cin:0})]
      node T_11 = or(a_and_b, b_and_cin) @[FullAdder.scala 33:23,
SECTAGS:{T_11:2})]
     node T_12 = or(T_11, a_and_cin) @[FullAdder.scala 33:35,
SECTAGS:{T_12:2})]
     io.cout <= T_12 @[FullAdder.scala 33:11]
```

Figure 6: Security label propagation.

The security labels are computed using the join operation on the lattice elements. However, we can also set security labels to specific, ignoring the default behavior. This can be done for declassification, for example. Example is given in Figure 7.

```
class FullAdder extends Module {
   val io = new Bundle {
      val a = UInt (INPUT, 1, NONE , ”0”)
      val b = UInt (INPUT, 1, NONE, “2”)
      val cin = UInt (INPUT, 1, NONE, “0”)
      val sum = UInt (OUTPUT, 1, NONE, “1”)
      val cout = UInt (OUTPUT, 1, NONE, “0”)
     }
     //Generate the sum
     val a_xor_b = io.a ^ io.b setSecTag(“3”)
     io.sum := a_xor_b ^ io.cin setSecTag(“5”)

     //Generate the carry
     val a_and_b = io.a & io.b
     val b_and_cin = io.b & io.cin
     val a_and_cin = io.a & io.cin
     io.cout :=
     a_and_b | b_and_cin | a_and_cin
}
```

```
circuit FullAdder :
   module FullAdder :
      input clk : Clock
      input reset : UInt<1>
      output io : {a : UInt<1>, b : UInt<1>, cin : UInt<1>, sum : UInt<1>,
cout : UInt<1>} @[SECTAGS:{cout:0,sum:1,cin:0,b:2,a:0}]

      io is invalid
      node a_xor_b = xor(io.a, io.b) @[FullAdder.scala 24:23,
SECTAGS:{a_xor_b:3})]
      node T_10 = xor(a_xor_b, io.cin) @[FullAdder.scala 26:22,
SECTAGS:{T_10:5})]
      io.sum <= T_10 @[FullAdder.scala 26:10]
      node a_and_b = and(io.a, io.b) @[FullAdder.scala 30:23,
SECTAGS:{a_and_b:2})]
      node b_and_cin = and(io.b, io.cin) @[FullAdder.scala 31:25,
SECTAGS:{b_and_cin:2})]
      node a_and_cin = and(io.a, io.cin) @[FullAdder.scala 32:25,
SECTAGS:{a_and_cin:0})]
      node T_11 = or(a_and_b, b_and_cin) @[FullAdder.scala 33:23,
SECTAGS:{T_11:2})]
     node T_12 = or(T_11, a_and_cin) @[FullAdder.scala 33:35,
SECTAGS:{T_12:2})]
     io.cout <= T_12 @[FullAdder.scala 33:11]
```

Figure 7: Overwriting security label value during propagation.

Composite datatypes, e.g. Reg and Wire, are constructed upon basic datatypes like UInt. Security label definitions for the basic type can then be used for the composite data types.
After generating the FIRRTL representation, it is parsed to generate a FIRRTL tree file which has Z3 format. The file contains the assertions generated based on the input Chisel code, as defined by designer and calculated by security label propagation rules. The propagation rules can use join or meet of multiple security labels

when computing the output's security label. The whole flow from Chisel to Z3 is shown in Figure 8.

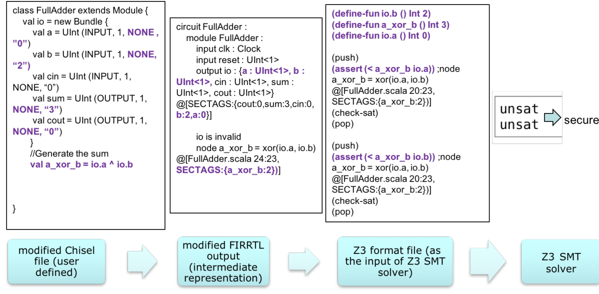

Figure 8: Parsing to a FIRRTL tree file.

3.2 Specifying Lattice for the Security Labels

We use lattice structure to construct and define our security labels. A map in Chisel is used to define the lattice structure, and it is propagated to Z3. Figure 9 shows a concrete example of how lattice-based security hierarchy is defined and used in the Chisel file, and how it parsed to FIRRTL and Z3.

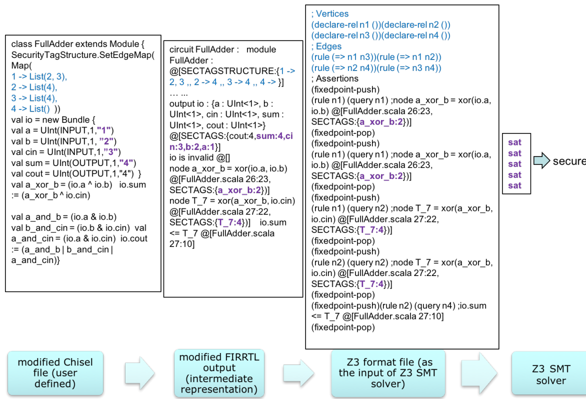

Figure 9: Specifying lattice structure in Chisel and passing it to Z3 for use during verification.

4. DYNAMIC SECURITY LABELS

Static security labels can help enforce the information flow policy, however, they does not allow any shared resources across security levels. Yet, it is possible to achieve resource sharing across security levels in hardware design without reducing the security guarantees. To verify this kind of design, we introduce dynamic security labels. While our verification remains at the design time, the security label can be dynamically dependent on variables in hardware design. For example, in a data port, the security label of data might depends on the address of that data. In this case, with dynamic labelling, the label of the data can be defined as a function of the address. Then, the data port can be shared by all the security levels. But at each possible time, it can be checked that the data port is used for only one security level. The verification will enforce the information flow policy in the resource sharing situation with the dynamic labels.

4.1 Dynamic Label Specification

These are two conditions to construct dynamic labels in SecChisel. On the one hand, there exist possibilities that a variable's security label may depends on our variable's value or type. On the other hand, if we consider nested modules, security labels can be even brought from other places and requirements may be needed to translate variable's security label between different modules' interference tables (interference table is discussed in Section 5).

For example, for the first condition, we can assume we define dynamic labels in this pattern:

```
val a = UInt (INPUT, 1, "L");
- for static label
val b = UInt (INPUT, 1, "SecMap(a)");
- for dynamic label, based on value of variable
a in current module
```

Where SecMap() is a map function which take variable a's value as input and output mapping result as variable b's security tag. For the case of security tags coming from other modules, we can define dynamic labels as:

```
val c = UInt (INPUT, 1, "SecMap(ModuleName:a)");
- for dynamic label, based on value of variable
a in another module, ModuleName
```

4.2 Generating Z3 Statements for Dynamic Labels

Dynamic labels are transferred through FIRRTL similar to static labels. The main difference is in their correspondent SMT statements. All of the possible security labels that a dynamically labeled variable can have by its mapping function is tried by SMT solver to see if any of the possible values creates an information flow violation. However, possibility set is reduced by context of the circuit at that point. If that point of the circuit is under any kind of branching that can only be triggered by certain values for dynamically labeled variable (statements inside of an if statement, for example), SMT solver does not try all possibilities but the ones that satisfies branching condition. This reduction of possible security label space enables usage verification in case of shared resources through a controlled environment. Such optimisation, however, becomes difficult in case of nested modules where the security label is based on variable in another module, then all possible values of that variable at any time need to be considered.

5. NESTED MODULES

One of the difficulties in hardware verification is handling of modules. Especially, one of the problems of using modules during the verification is that the modules might be used many times with different parameters

with different security labels. Hence in each usage, we need to determine the security labels separately. Furthermore, it would not be difficult if all the modules in a design share the same lattice structure. However, modules can be developed by different developers, and provided with different lattice structures. Therefore, compatibility of lattice structures is an issue. Another problem is dealing with blackbox modules. Sometimes developers may use modules where there is no information about the inner details of the modules. Therefore it may not be possible to deduct how input and output wires are interacting unless additional information is give (or the module is reverse-engineered). For SecChisel design, we present two alternative methodologies that can be used in the security verification of design with nested modules.

5.1 Lattice Reduction

In the first approach, we simply ignore the lattice structure of the inner module. The outer module's lattice can be used for inner module by removing labels of all the internal variables of the inner module. The input wires will be labeled depending on the labels of the outputs of the outer module's wires that connect to the inner module. All the remaining labels will by dynamically deducted based on their relations with the input and output wires. After each output signal receives its label, they will be compared with the expected output labels of the outer module. The outer module will also supply expected labels of the output wires.

For confidentiality, if the resulting read security labels of an output is less than or equal to expected read security label than it will be accepted otherwise it will not be accepted. For example, the module shown in Figure 10 for inputs a, b, and c with security levels 1, 2, and 3 respectively. The labels of the k, l, m, and n are calculated based on the labels of a, b, and c. The labels of outputs wires k, l, m, and n will have label values 2, 3, 2, and 2 respectively. Note that this calculation is done using the outer module's lattice. The success of verification depends on the compatibility of actual and expected labels of the output wires. For example, if the expected values of labels for k, l, m, and n were 1, 4, 1, and 3, the verification will fail. Since the actual value of l and n are greater than its expected values as in Figure 10.

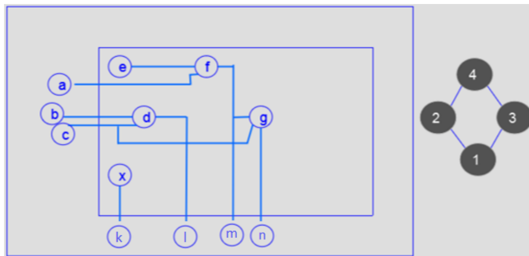

Figure 10: Example of nested module.

5.2 Interference Table

The second alternative is to use a table for representing which output wire depends on which input wire as in Table 1. Suppose we have three input wires A,B and C and three output wires, the intersection of these wires in the table represents a dependence. For example, in the table output wire X depends on A and C. This means we can deduct the security label of X based on the labels of A and C. We call this table as interference table. The table can be calculated before the verification. For blackbox modules, we can rely on the table given by a trusted developer. With the table, during the verification the labels of the output wires can be dynamically calculated based on the labels of input wires.

Table 1: Example Interference Table

	X	Y	Z
A	1	0	1
B	0	1	0
C	1	0	0

Both alternatives have their advantages and disadvantages. Lattice reduction does not need a pre-processing phase and does not require extra memory for storing tables. However, it may take longer to calculate the security labels of the output wires since inner module has to be analyzed for every single usage. Interference table is the opposite. It requires a prepossessing to calculate the interference table, and as the size of the input and output wires grow, it takes more and more memory. However, using interference table the labels of output wires can be calculated quickly.

6. OPTIMIZATIONS

The input of the verification process is labeled variables, instructions and the lattice. For small designs the execution time of the model checker (Z3 in our case) will not be an issue. However as the size of the input grows the processing time can be an issue as the designer may frequently modify the design in order to comply with security properties. Therefore, optimizing verification process to decrease the run-time is essential. Especially, the complexity of the SMT solving is the limiting factor in the size of the designs that can be checked. This project aims to incorporate a number of optimizations that span a number of heuristics, and more will be added.

The security labels are taken from the annotated Chisel code, passed to the FIRRTL representation, and finally input to SMT solver. Prior work has looked at translating directly from the hardware description language, e.g. Verilog, to SMT solver statements. This may result in excessive number of SMT solver statements. In general, model checkers find a solution by performing a search in a graph that represents the problem. Therefore, if we can prune the graph before the actual search, it decreases the overall verification time. The first optimization heuristic is to limit the search space by merging or ignoring some of the instructions, labeled variables or lattice values. In order to do this optimization a pre-processing is needed to see how this pruning can

be done. This pre-processing step can be performed exactly on the FIRRTL representation.
Second optimization can limit the label space by having a table that represents all possible labels. A naive Z3 representation results in possibly large number of implied security labels, depending on how the lattice structure is coded; resulting in increased runtime.
Third, the search space of dynamic labels can be limited. For dynamic labels referencing current module, the code structure of the module can be used to limit the possible values for the reference variables, and thus the possible labels values. This, for example, can be done for block of code inside an `if` statement, where the statement limits possible values of the control variable, and thus the possible values of the dynamic labels. For dynamic labels referencing external modules, such optimisation is not possible. Nevertheless, a table can be built of all the possible values of the reference variable over all executions. In certain cases, the reference variable may have few or only 1 value, e.g. a register is assigned a fixed value, and dynamic label uses that register – thus there is only one possible value of the security label, corresponding to the fixed value of the referenced variable.
Further optimizations are possible, and will explored by this project.

7. DESIGN STATUS

Currently, SecChisel is implemented using Chisel 3 and latest version of Z3 SMT solver. The static labels are implemented and can be verified for single module designs in Z3. Active development is under way for dynamic labels discussed in Section 4 and nested modules discussed in Section 5. The syntax of SecChisel is also under active development and may change from current one presented in the examples. Furthermore, this project aims to improve SecChisel with the following three expansions.

7.1 Expected SecChisel Expansions

Our work will integrate the optimizations mentioned Section 6. These, and possibly others, will be added to reduce the runtime, and allow for designs with multiple nested modules to be realized and verified in an efficient manner.
Current work focuses on designs where all modules have security tags taken from same lattice. The problem of different lattices will be addressed and incorporated into SecChisel to allow designers to work with 3rd party modules, making the framework more practical.
Major goal of the ongoing work is on allowing for timing channel analysis as well, and the work will add temporal labels. Target is verify complex designs, such as caches. Temporal labels will also help with implicit flows, as currently SecChisel only considers explicit flows.

8. RELATED WORK

A small number of projects have explored security verification of hardware architectures, and they are listed below. Our more detailed survey of hardware and software verification methodologies [6] has extended details about these, and other, projects.
VeriCoq is a tool that provides mechanisms to transform Verilog code into code with PCHIP (Proof-Carrying Hardware Intellectual Property), which makes it possible for the customers to verify the security of the design written in Verilog. In addition, the newer version of VeriCoq also supports the verification of the information flow property [3].
Caisson [16] is a hardware description language with static information flow verification at design time. Using Caisson, the authors were able to create the first provably information-flow secure processor that contains a time-multiplexed pipeline and a partitioned cache. Their pipeline is secured by secure time multiplexing (or time lease) with separated context (i.e., registers and memory) for each security level. The cache is statically partitioned between different security levels.
Sapper [15] is another hardware description language that is based on a synthesizable subset of Verilog. Sapper compiler automatically enforces non-interference in the generated hardware logic. Authors of Sapper realized a processor that was designed and simulated in ModelSim [9]. Unlike Caisson [16], data with different security labels can share resources in Sapper, e.g. registers, resulting in lower overheads.
SecVerilog [26] is Verilog extended with information flow annotations. It enables static checking of hardware information flow. Unlike Sapper, which uses dynamic information flow at runtime, information flow checking in SecVerilog is done during compile time which provides a better run time performance.
Unlike Verilog-based work, XOM architecture was formally specified and then verified [17] against an adversary in Murφ model checker. The eXecute Only Memory [18] (XOM) is a hardware design with embedded cryptographic functionality and access control. By adding new hardware and new instructions, XOM is able to protect user data from a malicious operating system. On-chip data is isolated using hardware tags labeling the owner of data, while off-chip data is protected by encryption and hashing.
In a different approach, Micro Policies work presents a generic approach for formalizing and verifying IFT policies. It is based on recent work on Programmable Unit for Metadata Processing (PUMP) [7], which added programable metadata processing unit alongside with the data computation. PUMP allows programmers to create policies and rules that enforce IFT mechanisms by manipulating the metadata tags in each instruction that a processor executes.
Another type of work is Cache Verification [27] that focuses on understanding side-channel leakage and creating models based on the non-interference property between an attacker and a victim process that are using same processor cache. Zhang and Lee used Murφ to enumerate all possible states and transitions, and count the number of inferences between attacker and victim for the different state transitions. Based on this data,

mutual information [5] is then used to quantitatively analyze the interference between the two processes, and reveal side-channel vulnerabilities.

9. CONCLUSION

SecChisel project develops a security verification methodology that is applicable to different hardware-software security architectures during the design phase. The verification framework aims to prove that a system holds desired properties with respect to confidentiality and integrity. SecChisel focuses on information flow and non-interference properties for verification. Using these properties, confidentiality and integrity of the sensitive data can be checked at design time. The proposed verification framework is built upon Chisel hardware construction language, and leverages Z3 SMT solver. Present design integrates static labels and allows for checking designs without nested modules. Ongoing work focuses on dynamic labels, temporal labels, and support for nested modules. Furthermore, number of SMT solver optimisations need to be added to ensure efficient runtime for large designs.

10. ACKNOWLEDGEMENT

This work is supported in part by the National Science Foundation (NSF) grants 1524680; and Semiconductor Research Corporation (SRC) contract 2015-TS-2633. Dr. Demir's work is also supported in part by TUBITAK grant 2219 and TUBITAK grant 1059B191401391.

References

[1] AMD. AMD Memory Encryption, 2016. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, accessed May 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[3] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. Vericoq: A verilog-to-coq converter for proof-carrying hardware automation. In *International Symposium on Circuits and Systems (ISCAS)*, pages 29–32. IEEE, 2015.

[4] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[5] Thomas M Cover and Joy A Thomas. *Elements of information theory.* John Wiley & Sons, 2012.

[6] Onur Demir, Wenjie Xiong, Faisal Zaghloul, and Jakub Szefer. Survey of approaches for security verification of hardware/software systems. Cryptology ePrint Archive, Report 2016/846, 2016. `http://eprint.iacr.org/2016/846`.

[7] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. Pump: a programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.

[8] Jeffrey S Dwoskin and Ruby B Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 389–400. ACM, 2007.

[9] Uwe Hatnik and Sven Altmann. Using modelsim, matlab/simulink and ns for simulation of distributed systems. In *International Conference on Parallel Computing in Electrical Engineering*, pages 114–119. IEEE, 2004.

[10] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. *ACM SIGPLAN Notices*, 50(4):517–529, 2015.

[11] 4th gen core family desktop specification update. 2016. `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/4th-gen-core-family-desktop-specification-update.pdf`.

[12] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 272–283. ACM, 2011.

[13] Taeho Kgil, Laura Falk, and Trevor Mudge. Chiplock: support for secure microarchitectures. *ACM SIGARCH Computer Architecture News*, 33(1):134–143, 2005.

[14] Ruby B Lee, Peter Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 2–13. IEEE, 2005.

[15] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. Sapper: A language for hardware-level security policy enforcement. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 97–112. ACM, 2014.

[16] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Notices*, volume 46, pages 109–120. ACM, 2011.

[17] David Lie, John Mitchell, Chandramohan A Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *Symposium on Security and Privacy*, pages 166–177. IEEE, 2003.

[18] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[19] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

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

[21] Bruce Schneier. The internet of things will turn large-scale hacks into real world disaster, 2016. `https://motherboard.vice.com/read/the-internet-of-things-will-cause-the-first-ever-large-scale-internet-disaster`, accessed July 25, 2016.

[22] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM, 2003.

[23] Jakub Szefer and Ruby B Lee. Architectural support for hypervisor-secure virtualization. In *ACM SIGPLAN Notices*, volume 47, pages 437–450. ACM, 2012.

[24] ARM Trustzone. Trustzone information page. Technical report, 2016.

[25] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st annual international symposium on Computer architecuture*, pages 457–468. IEEE Press, 2014.

[26] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 503–516. ACM, 2015.

[27] Tianwei Zhang and Ruby B Lee. New models of cache architectures characterizing information leakage from cache side channels. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 96–105. ACM, 2014.