SecChisel Framework for Security Verification of Secure Processor Architectures

Shuwen Deng[1], Doğuhan Gümüşoğlu[2], Wenjie Xiong[1], Sercan Sari[3], Y. Serhan Gener[3], Corine Lu[1], Onur Demir[3], and Jakub Szefer[1]
[1]{shuwen.deng, wenjie.xiong, corine.lu, jakub.szefer}@yale.edu,
[2]doguhan.gumusoglu@std.yeditepe.edu.tr, [3]{ssari, sgener, odemir}@cse.yeditepe.edu.tr
[1]Yale University, New Haven, CT 06510, USA
[2,3]Yeditepe Üniversitesi, 34755 Ataşehir/İstanbul, Turkey

ABSTRACT

This work presents a design-time security verification framework for secure processor architectures. Our new *SecChisel* framework is built upon the Chisel hardware construction language and tools, and uses information flow analysis to verify the security properties of an architecture at design-time. To enforce information flow security, the framework supports adding security tags to wires, registers, modules, and other parts of the design description, as well as allows for defining a custom security lattice and custom information flow policies. The framework performs automatic security tag propagation analysis in a new SecChisel parser and information flow checking using the Z3 SMT solver. The same SecChisel codebase is used to design hardware modules as well as to verify the security properties, ensuring that the verified design directly corresponds to the actual design. This framework is evaluated on RISC-V Rocket Chip expanded with AES and SHA modules. The framework was able to capture information leaks in the hardware bugs or Trojans that it was tested with.

CCS CONCEPTS

• **Security and privacy → Logic and verification**; **Embedded systems security**; **Information flow control**; • **Hardware →** Hardware description languages and compilation.

KEYWORDS

formal security verification, secure processors, Chisel, RISC-V

ACM Reference Format:
Shuwen Deng[1], Doğuhan Gümüşoğlu[2], Wenjie Xiong[1], Sercan Sari[3], Y. Serhan Gener[3], Corine Lu[1], Onur Demir[3], and Jakub Szefer[1]. 2019. SecChisel Framework for Security Verification of Secure Processor Architectures. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3337167.3337174

1 INTRODUCTION

A number of secure processor architectures have been designed over the last decade, e.g., XOM [15], AEGIS [19], SP [11], Bastion [8], or HyperWall [20]. They all implemented some new security protection mechanisms in hardware, typically leveraging encryption and hashing to protect user's code or data. These new mechanisms are used, for example, to enable isolation of trusted software modules from an untrusted operating system or to protect virtual machines from an untrusted hypervisor, for example. The ideas presented by these architectures have been recently incorporated into commercial designs, such as AMD's SEV [1] or Intel's SGX [16] processor architecture security extensions.

Today, however, most of these architectures have not been thoroughly and formally verified from the security perspective. Any security flaws with the hardware will undermine the security of the whole platform. There is then a need for security verification frameworks, such as presented in this paper.

To help performing security verification of such architectures, this paper proposes the new *SecChisel* framework, which incorporates security-related features directly into Chisel [4] code. Similar to existing projects, Caisson [14] and SecVerilog [23], SecChisel embeds security tags in a modified hardware description language. New security tags are used in information flow tracking approach at design-time and do not add overhead to the actual hardware. The new methodology supports dynamically valued tags, not available in Caisson [14], which is useful when reasoning about primitives (such as wires or registers) that may hold public or private data during different cycles of computation. And, unlike SecVerilog [23], users of SecChisel need not learn nor write SMT code. To the best of the authors' knowledge, this is also the first hardware security verification framework supporting nested modules, without having to check individual module separately. Projects focusing on run-time checks, such as Sapper [13], are orthogonal to this work.

To demonstrate how SecChisel can protect secure processor architectures, the framework is validated on the Rocket Chip [3] RISC-V [18] processor, developed in Chisel, by implementing and verifying AES and SHA security modules realized as Rocket Custom Coprocessor Interface (RoCC) accelerators within the RISC-V processor. In addition, synthetic hardware bugs and Trojans are introduced into the AES and SHA modules to show how the framework can detect information leaks. The security verification of the accelerators takes an average of 21 minutes, while the compilation and simulation of the Rocket Chip with RoCC takes about 25 minutes. This shows that the verification step can be done in parallel with the compilation of the design and does not introduce new timing overhead.

1.1 Contributions

The contributions of this paper are:

- The first security verification framework based on the Chisel hardware construction language, which leverages information flow tracking and an SMT solver.
- The first hardware security verification framework supporting verification of nested modules, without having to check individual module separately.
- Flexible security verification framework supporting static and dynamic tags, declassification mechanisms, and interference tables for third party modules.
- Evaluation of the framework's functionality and performance using AES and SHA security RoCCs within a RISC-V Rocket Chip, showing fast runtime and the ability to detect information leaks due to hardware bugs or Trojans.

2 VERIFICATION METHODOLOGY

The goal of this work is to provide a design-time methodology to formally prove security properties of a secure processor architecture (this Section) and a practical framework using the methodology (Section 3). Works on run-time security checks, e.g., Sapper [13] or GLIFT [21], are complementary to this work.

2.1 Assumptions and Threat Model

Either through a bug in a hardware code, or due to a malicious designer or an adversary, some sensitive data may leak out to untrusted low-security outputs, a so-called "information leak." The framework checks, at design-time, if there are any such buggy or malicious flows of information. Using the information flow tracking, policy violations such as confidentiality violation or integrity violation can be detected. Information leaks via physical channels, such as EM radiation, are not considered as they cannot be expressed in today's hardware description languages. Hardware bugs or Trojans at design time are considered, but after-manufacturing bugs [7] are orthogonal to this work. The framework assumes a trusted compiler and toolchains to convert Chisel to an HDL and then the actual hardware.

2.2 Information Flow Tracking Approach

Our work uses information flow tracking (IFT) approach. Information flow refers to the transfer of information between different entities. Information flow can be explicit, e.g., $a = b$; where data or information in b goes to a; or it can be implicit, e.g., $b = 0$; if (a) then $b = 1$; where the value of b reflects whether a is true, but there is not a direct assignment, or copying of data, from a to b. Typically, when discussing information flow there are different security levels, e.g., a lower-security level ("Low") such as public data, a higher-security level ("High") such as secret key. Each data is associated with a security level, and information flow tracking can be used to check the security properties, e.g., no transfer of "High" to "Low" information (for confidentiality) or "Low" to "High" information (for integrity). Since information flow tracking has inherent presence of false positive, the SecChisel framework supports tagging at very fine granularity (individual bits) and using declassification and dynamic tags to minimize the false positives.

3 THE SECCHISEL FRAMEWORK

The proposed methodology is realized in a new SecChisel framework. The framework extends the existing Chisel language and tools with new security verification functionality. The SecChisel workflow is shown in Figure 1. SecChisel extends data variables (e.g., various wires, registers, or other parts of the design) of Chisel with security tags, allowing designers to annotate the design with the security tags associated with variables. During compilation, the SecChisel code is converted to a modified FIRRTL (Flexible Intermediate Representation for RTL) [12] and then translated to logical statements that can be used with the Z3 SMT solver [9], which checks for information flow violations based on the security tags. The SMT solver is used to assert that there are no data transfers between variables that could violate the security policy. The security verification steps can be done in parallel with compilation and simulation of a Chisel design. The whole SecChisel workflow consists of:

(1) **SecChisel Code** – hardware description, including the security lattice description, the new security tags, the dynamic tag-range functions, and declassification.
(2) **SecChisel Parser** – tool for generating the modified FIRRTL that contains both functional description of the design and information about the security tags.
(3) **FIRRTL Code** – intermediate representation of the design with information about the security tags.
(4) **SMT Code Generator** – tool for parsing FIRRTL into a FIRRTL statement/expression tree, which is then processed into SMT statements used by an SMT solver.
(5) **SMT Code** – code describing the security lattice, the tags, the dynamic tag-range functions, data flows, and the assertions for information flow checking.
(6) **Parallelization** – tool that parallelizes the SMT code according to the number of processor cores available.
(7) **Z3 SMT Solver** – tool that does the actual information flow checking and generates satisfiable or unsatisfiable result from the SMT code.
(8) **Interference Table** – an optional step where third-party black-box modules can be used as part of the verification.

3.1 SecChisel Code (① in Figure 1)

SecCoreModule class extension and security policy. There is a new `SecCoreModule` class that extends the `CoreModule` class from Chisel and allows modules to have security lattices bound to them. Figure 2a shows the sample SecChisel code of a SHA-256 engine realized as a Rocket Chip RoCC. The *io.addend* and *io.accum* are input ports. The module is extended from the new base module `SecCoreModule` to allow its components to be tagged (line 1), i.e., each basic variable can be associated with a security tag for information flow analysis. All variables that are not tagged or within normal `CoreModule` modules have their security tags set to undefined by default. Undefined tags are resolved in the SMT Code Generator step.

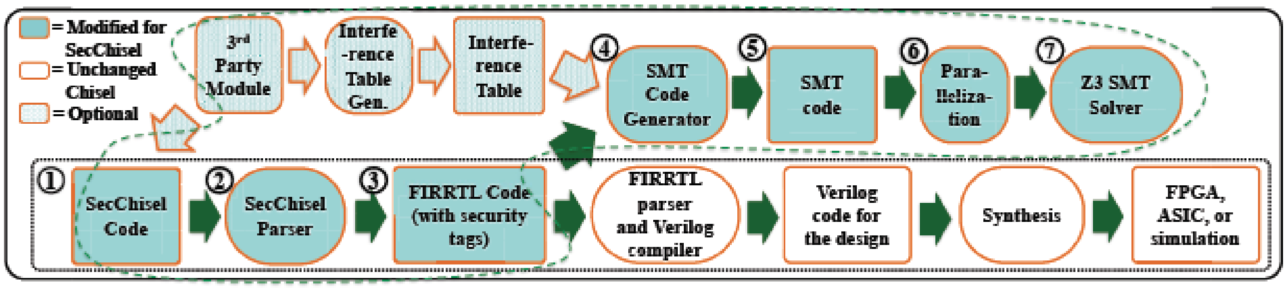


Figure 1: SecChisel verification workflow. Square boxes represent files or data, ovals represent tools or processes. The unmodified Chisel tools (in white) can be used to generate the hardware design, while the new SecChisel components (in turquoise blue) perform the security verification. Black dotted line circles pre-existing baseline Chisel. Green dashed line includes whole SecChisel verification flow. Because the security tags are embedded in the source code of the design, single codebase can be used for both security verification as well as to generate the hardware design. The use of third-party modules and interference table is optional in addition to help support third-party IP.

The default security policy is for enforcing confidentiality: it is not allowed that variables bound to higher security tags leak their data to the variables with lower security tags. The policy is checked in Z3 SMT Solver, thus does not require additional specification. Integrity can be verified in a similar manner.

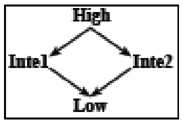
Security lattice definition. The base `Lattice` class contains the simplest possible security lattice with two levels: "High" and "Low", where "High" has greater security level than "Low". Any new security lattice can be created by extending a new object with the base `Lattice` class. Any two tags' values in the security lattice have a greatest lower bound (the meet) and least upper bound (the join). The meet or join operations are used to calculate the resulting tag value when variables are processed (Case 2 of Section 3.4).

For each `SecCoreModule`, the designer can define the module's own security lattice or implicitly use the default `Lattice` class. Sub-modules are allowed to have different security lattices rather than use top module's security lattice. Within the object, new security lattice elements are defined, and the relationship among the elements is defined using the overloaded less-than $<$ operator to show security tag relations between each other. Figure 2a shows an example of a designer-defined new security lattice (lines 14 - 19) including a visualization of the security lattice for that class. The custom lattice will overwrite the default one (line 12).

Static tags. Static tags of variables in the design do not change their values throughout the verification process and always have the fixed value assigned by the designer. They are used when the designer is certain about a variable's security level for the whole life cycle of the system. In line 5 of Figure 2b, which shows SecChisel code for an AES engine realized as a Rocket Chip RoCC, the encryption key of AES RoCC is tagged as "High" security using the :> operator.

Dynamic tags. Dynamic tags are tags of which the value depends on the data value of other variables (wires, registers, etc.), which are known as the dependent variables. The value of the dynamic tag is a function of the dependent variables. The function outputs one of the values from the security lattice, based on rules specified by the designer, called "tag-range functions." A tag-range function for dynamic tags in SHA RoCC is defined in lines 21 - 24 of Figure 2a. The dynamic tags of *message_input* and *message_output* is determined by *process_id*, and is assigned by the overloaded :> operator (line 26 - 30 of Figure 2a) combined with the usage of

```
 1. class SHA256_init (implicit p: Parameters)
                                  extends SecCoreModule()(p){
 2.  val io = IO (new Bundle {
 3.    val addend = UInt(64.W).asInput
 4.    val accum = UInt(64.W).asInput
 5.    val initReady = Bool()
 6.    val message_output = Vec(16, UInt(32.W)).asOutput})
 7.  val message_size = io.accum(SIZE_MSB,SIZE_LSB)
 8.  val process_id = io.accum(ID_MSB,ID_LSB)
 9.  val message_in = Cat(io.addend(M_PART_ONE_MSB,
      M_PART_ONE_LSB),io.accum(M_PART_TWO_MSB,M_PART_TWO_LSB))
10.
11.  // Specify use of custom lattice
12.  override val lattice = SHA_Lattice
13.
14.  // Define lattice structure
15.  object SHA_Lattice extends Lattice {
16.   val Inte1 = NewLatticeElement()
17.   val Inte2 = NewLatticeElement()
18.   LOW < Inte1 < HIGH
19.   LOW < Inte2 < HIGH }
20.
21.  // Dynamic value tag function
22.  val SHA_TagRange =
23.   createTagRange(lattice.LOW).add(0, 130, lattice.Inte1)
24.  .add(131, 500, lattice.HIGH).add(501,1000,lattice.Inte2)
25.
26.  // Dynamic value tags
27.  message_in :> (SHA_TagRange, process_id)
28.
29.  for (i <- 0 to 16-1){
30.   io.message_output(i) :> (SHA_TagRange, process_id)}
31.  …
32. }
```



(a) Example from SHA RISC-V Rocket Chip RoCC with custom security lattice shown in the sqaure box.

```
1. class KeyExpansion(implicit p: Parameters)
                                extends SecCoreModule()(p){
2.  …
3.  val roundkey = Mem(16, UInt(width = 8))
4.  roundkey(0):= "h2b".U; …
5.  roundkey :> lattice.HIGH
6.  …
7. for(i<-0 to 176-1){
8.   io.data_output(i) :> (roundkey_propagate, lattice.LOW)}
```

(b) Example from AES RISC-V Rocket Chip RoCC.

Figure 2: Example from SHA and AES RISC-V Rocket Chip RoCC written in SecChisel code, new additions compared to base Chisel code are in bold. For the Chisel code not in bold, please refer to Chisel specification [4]. A custom security lattice is shown, of which "Inte1" and "Inte2" are extra security tags that have security level between "High" and "Low".

tag-range function. In this case, the tag of the input message and output hash value will be either "High" or "Low" determined by *process_id* and all of the undefined tags in the sub-modules will resolve to have the same dynamic tags. The tag-range function can also be made to support multiple dependent variables by defining sets of ranges.

Declassification. Information flow will always report a violation if there is information flow from "High" to "Low" (for confidentiality). Sometimes, however, this kind of information flow should be allowed. For example, in lines 7 - 8 of Figure 2b, the final output *data_output* of AES RoCC is declassified to be "Low" using :> operator. Although the output depends on the secret key and will be tagged as "High", since strong encryption is assumed, the output conveys no information to the attacker, and thus, it can be *declassified* to "Low" value.

When using declassification, the system designer might overwrite some rules and use declassification improperly, causing a false negative. Our tool reports the number of declassifications used (to allow users to compare it with the expected number) and also gives warnings about declassification.

Nested modules. Chisel and SecChisel both support describing a design with nested modules. To analyze the information flow, the nested modules will be resolved in SMT Code Generator described in Section 3.4.

3.2 SecChisel Parser (② in Figure 1)

Given SecChisel code, it needs to be parsed into the modified FIRRTL code. The parser is based on Chisel parser, but it includes the tags information for the variables (especially, it marks untagged variables as undefined). Moreover, Chisel sometimes transparently defines new temporary variables during compilation which are not in the original Chisel source code. These will be tagged as undefined in the modified FIRRTL code as the parser generates them.

3.3 FIRRTL Code (③ in Figure 1)

FIRRTL language was created to represent the standardized, elaborated circuit produced from Chisel code [12]. It can be efficiently used to analyze the information flow. SecChisel does not modify FIRRTL language's syntax. Instead, the security information is embedded in the comments section of each line of FIRRTL code. When analyzing the FIRRTL, Chisel's default tools will ignore the comments so that the hardware can be generated directly from the SecChisel code without any changes to the back-end of the Chisel tool-chain. Meanwhile, when FIRRTL is analyzed by the SMT Code Generator, the security information is included to generate the SMT code. So the verification and the final hardware are based on the same design in FIRRTL.

3.4 SMT Code Generation (④ in Figure 1)

SMT Code Generator converts FIRRTL into an expression/statement tree and then generates SMT code. Processing the FIRRTL tree to SMT statements is the key part of the SecChisel framework, especially when dealing with nested modules. The four phases for transforming FIRRTL code into SMT code are:

(Phase 1) Parse the FIRRTL file and create $L_{taggedVariable}$ structure to store variables and the corresponding explicit tag information in the structure, untagged variables will have no tags associated with them yet.

(Phase 2) Create tags for all variables: apart from variables explicitly tagged by the designer in SecChisel, variables with no tags are tagged with $UndefinedTag$, and all data is stored in new $L_{default}$ structure.

(Phase 3) Resolve all undefined tags in $L_{default}$ through nested modules of the circuit and store the data in the $L_{redefine}$ structure.

(Phase 4) Output SMT code, F_{SMT}, based on security lattice, tag-range functions and tag information in $L_{redefine}$.

Algorithm 1 *redefineTags* $(statement, variable, curModule)$

```
Input: $statement$: a line of FIRRTL code containing the variable of
    $L_{default}$, whose tag needs to be redefined
    $variable$: the variable (e.g., Reg, Wire, etc.) whose tag needs to be
    redefined
    $curModule$: the module that is being checked
Output: redefined tag for $variable$ of $L_{redefine}$
 1: if $variable$ has been assigned defined tag then
 2:     return defined tag
 3: else
 4:     for each statement $x \in L_{default}$ of $curModule$ do
 5:         if $x$.lhs == $variable$ then
 6:             if tag of $x$.lhs is defined (or tag of $x$.rhs is defined) then
 7:                 tag of $statement$.rhs ⇐ tag of $x$.lhs (⇐ tag of $x$.rhs)
 8:                 return tag of $statement$.rhs
 9:             else
10:                 find submoduleList of current module
11:                 tag of $statement$.rhs ⇐ joinRedefineTags ($statement$, rhs,
                    submoduleList, $curModule$)
12:                 tag of $x$.rhs ⇐ tag of $statement$.rhs
13:                 tag of $x$.lhs ⇐ tag of $x$.rhs
14:                 return tag of $statement$.rhs
15: if cannot find $statement$.rhs then
16:     if $curModule$ has outer module then
17:         tag of $statement$.rhs ⇐ redefineTags ($statement$,
            $statement$.rhs, outer module)
18:         return tag of $statement$.rhs
19:     else
20:         tag of $statement$.rhs ⇐ lowest tag of its security lattice
21:         return tag of $statement$.rhs
```

In Phase 1, data structure $L_{taggedVariable}$ is generated to store security information derived from FIRRTL file. The data structure $L_{taggedVariable}$ contains the variables and their tags' information: "statically tagged variable" has explicit security tag defined by the system designer, "dynamically tagged variable" has tag-range function and the dependent variable(s) defined by the system designer, and "untagged variable" does not have any tags assigned, i.e., such variables have no defined tags in the SecChisel code.

In Phase 2, tags for variables associated with the left-hand side (lhs), or the right-hand side (rhs), of statements are created based on information from variables already tagged in $L_{taggedVariable}$. After this phase, all the tag information will be stored in $L_{default}$ structure. Especially, there are seven types of FIRRTL statements. In order to simplify tag assignment, these seven types of FIRRTL statements can be classified into the following three cases:

(Case 1) Definitions: a variable is defined to be a constant, e.g., $a = 12$.
(Case 2) Assignments: a variable is assigned the results of some operations of other variables, e.g., $b = c + d$.
(Case 3) Connections: a variable is assigned to have the same value as a different variable, e.g., $e = f$.

For Case 1 and Case 3, if the tag already has an static or dynamic tag value assigned, the tag will be kept; otherwise, the $UndefinedTag$ value will be assigned to the tag. Exceptionally, port variables of top modules without tags are assigned to the default lowest security level to guarantee no information will implicitly leak outside the circuit. For Assignments (Case 2), rather than generate a specific static or dynamic tag, a join statement following three ResRules defined next using tags of the rhs variables in the statement is generated. The tag resolution rules (ResRule) for statements of variables A and B on the right-hand side are:

(ResRule 1) $(Tag_A, Tag_B) \Rightarrow (join\ Tag_A\ Tag_B)$
(ResRule 2) $(Tag_A, UndefinedTag_B) \Rightarrow UndefinedTag$
(ResRule 3) $(UndefinedTag_A, UndefinedTag_B) \Rightarrow UndefinedTag$

Here the $(join\ Tag_A\ Tag_B)$ does not compute the join, but is an SMT statement that will be evaluated in the SMT solver. The Tag_A or Tag_B could end up being resolved as join of some other tags following tag relations defined in security lattice, so all the join operations are computed in the SMT solver at the very end. They can be either static or dynamic.

In Phase 3, $L_{default}$ will be used to generate list $L_{redefine}$, where all the $UndefinedTag$s stored in list $L_{default}$ will be checked recursively using $Algorithm$ 1 to go through nested modules until there is an assignment statement that assigns some defined tag to the currently $UndefinedTag$; this can be a static tag, a dynamic tag, or a generated join statement following ResRules.

Nested modules support in SecChisel will resolve all the $Undefi-\ nedTag$ first in the current module and then in different sub-modules and outer modules. One variable can be referenced in different layers of module hierarchy. In addition, dynamic tags are resolved to support dependent variables which are in other parent modules or in sub-modules. Specifically, in "redefineTags" – $Algorithm$ 1, function "joinRedefineTags" will go through nest modules ("submoduleList") of the current module ("*curModule*"), find and calculate the tag of the variable using "redefineTags" function recursively. Recall that the top-most `SecCoreModule` must have its inputs and outputs explicitly tagged, so eventually, all variables that have some connection to input or output will be assigned a definite tag. Only variables unconnected to the rest of the circuit will remain with $UndefinedTag$, and these will be synthesized away anyway.

In Phase 4, the security lattice structure, tag-range functions and the $L_{redefined}$ are used to generate SMT format rules and assertions and output an SMT file F_{SMT}.

3.5 SMT Code (⑤ in Figure 1)

The SMT code file, F_{SMT}, contains the information flow assertions generated based on the FIRRTL code following previous steps. To enable the assertions to work, it also needs the security lattice and tag-range functions expressed as SMT statements. It contains tag propagation rules which use join of multiple security tags when computing the right-hand side variable's security tag of statements discussed in Section 3.4. In order to speed up verification in SMT solver, the SMT Code Generator pre-computes "*join*" results for the variable pairs in the security lattice. Therefore, SMT solver can directly fetch the result when "*join*" operation happens.

For dynamic tags, each tag-range function has a unique ID used to look up the function in the SMT code. SMT solver will enumerate all the possible output (tag) values that the tag-range function can generate for a dynamic tag, based on the dependent variable. Thus, a $join\ Tag_A\ Tag_B$ statement may resolve to many possible tag values, if either Tag_A or Tag_B are dynamic tags.

3.6 SMT Code Parallelization (⑥ in Figure 1)

The Z3 SMT solver does the actual verification. Because of the very structured nature of the SMT code the framework generates, it is possible to parallelize the assertion checking. For the SMT code there are two parts in each SMT file: 1) the predefined rules for the security tags, tag-ranges, security lattice, and 2) actual assertions used for checking every operation's information flow. The rules are needed for all assertions, but the assertions can be checked independently of other assertions. The SMT code can be then parallelized by converting source SMT file into *n* different files, where each file has the same rules, etc., but the assertions are evenly split into the *n* files. These *n* files can then be executed by parallel processes.

3.7 Z3 SMT Solver (⑦ in Figure 1)

The SMT file is used as input to the Z3 SMT solver, but other solvers could be used as long as they can parse the same SMT-lib syntax. The assertions check for violations of information flow policy, thus somewhat counter-intuitively, if an assertion is "satisfied" there is a violation of information flow policy. If an assertion is "unsatisfied" there is no violation of information flow policy. The goal is to have all assertions be "unsatisfied". E.g., Chisel code "c := io.a", assertions corresponded can be "(assert (< c io.a)) (check-sat)" to ensure that assigned variable "c" should not have lower security level than "io.a", in which case information will be leaked.

3.8 Interference Table

Sometimes the design will make use of black-box third-party modules. It may not be possible to directly analyze the information flow inside such modules (e.g., no source code is given). The trusted creator of the third-party module can generate an *interference table* which lists how the inputs interact with the outputs for the module, i.e., the information flow from inputs to outputs. The interference table can then be used for the designers to reason about information flow across black-box third-party modules included by the designer in his or her design. The interference table can be generated directly using SecChisel code and done in parallel with SecChisel Parser without influencing the main SecChisel flow. The table and FIRRTL can both be used as input to SMT Code Generator.

4 EVALUATION

SecChisel framework is implemented upon Chisel [4][1] and the system complexity of the new framework is shown in Table 1 for each of the core parts of SecChisel.

To evaluate the effectiveness and performance of SecChisel, an AES-128 and a SHA-256 accelerators were implemented as Rocket Custom Coprocessor Interface (RoCC) within the Rocket Chip [3] RISC-V processor. The functionality and interoperability of AES RoCC and SHA RoCC within the Rocket Chip were tested to ensure functional tests pass. The SecChisel framework can process the whole Rocket Chip as it can handle both `SecCoreModules` and the unmodified `CoreModules`. Since SecChisel is a superset of Chisel, most of the code of Rocket Chip is unmodified, only the two accelerators and corresponding sub-modules are written as `SecCoreModules`. We evaluate RoCC cores within a RISC-V core. Our framework works with the whole Rocket Chip and can find improper information flows due to bugs or hardware Trojans. The evaluation was done using a server with two Intel Xeon E5 CPUs (total of 24 processor cores) running at 2.90GHz, with 64GB of memory.

4.1 AES RoCC Implementation & Verification

Firstly, a full 10-round AES-128 was implemented as an RoCC of Rocket Chip. AES-128 RoCC encryption block diagram is shown in Figure 3a, note decryption process is symmetric to encryption. In our sample AES RoCC implementation, security lattice structure used can be seen in Figure 2a. In the first sub-module – *KeyExpansion* of encryption process, the encryption *Key* is bound to have tag “High”. Other variables in the AES RoCC are untagged.

Without use of declassification (“AES RoCC v1” in Table 2), running the whole verification of AES RoCC results in detection of a possible information leak, where the encrypted output is tagged “High” because of the interaction with the secret key, but connects to the “Low” output of the RoCC module. In order to remove this false positive, declassification is used (“AES RoCC v2” in Table 2). Especially, the encrypted output can be declassified from “High” to “Low”, because assuming AES is a cryptographically strong algorithm, the encrypted data cannot be used to learn the plaintext. Now, there will be no more violation of information flow policy and there will be no false positives. Verification results illustrated above are shown in Column “Formal Verification Result” of Table 2. The decryption module can be verified similarly.

In order to further prove the effectiveness of our framework, we insert three kinds of hardware bugs or hardware Trojans into AES RoCC (denoted as “HBT"), as shown in Figure 3 (b-d) and “AES RoCC v2 with HBT1/HBT2/HBT3” in Table 2. HBT1 outputs the key when a special input data trigger is sent to the AES RoCC. HBT2 inserts a register and a time counter inside AES RoCC and the key will be output when counter is added to some trigger value. HBT3 inserts a finite state machine inside AES RoCC: when a series of special input data triggers is processed by AES RoCC in a specific order, HBT3 will output the key. These HBTs all send “High” security key to the output. Table 2 shows that SecChisel is able to detect all of the HBTs as it finds information flows from “High” data to “Low” outputs of the modules.

[1]Commit id: bb12fe7 from Chisel repository at https://github.com/ucb-bar/chisel.

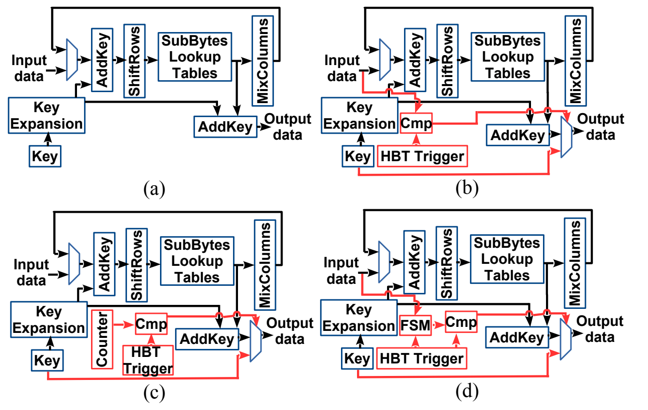


Figure 3: Block diagrams of AES-128 RoCC encryption modules without and with hardware bugs or Trojans. HBT components are shown in lighter color in the figures. (a) Basic AES encryption module. (b-d) Encryption module with HBT1, HBT2, and HBT3 hardware bugs or Trojans.

Table 1: System complexity of the SecChisel framework in terms of lines of code.

	Modified	Added	Total
SecChisel Parser	377	77	454
SMT Code Generator	4	2442	2446
Interference Table Generation	—	239	239
SMT Code Parallelization	—	28	28

4.2 SHA RoCC Implementation & Verification

SHA-256 RoCC was implemented in Rocket Chip to test static tags (“SHA RoCC v1” in Table 2) and dynamic tags (“SHA RoCC v2” in Table 2) of SecChisel. SHA-256 is a secure hash algorithm that is used to generate digests of messages to detect if the message has been changed. The inputs are message, message size and *process_id*. Initialization vector and other constants are hardcoded in the SHA-256 RoCC. The output is the hash value.

SHA can process both secret and public information (there is no secret key, just hash function). Therefore, if only using static tags for SHA RoCC, there are possibilities that input message is tagged “High” and output hash value is tagged “Low”, where false positives will be detected (Table 2). Using dynamic tags, the input and output of the SHA RoCC are both tagged with a dynamic tag, which depends on a *process_id* sent from input. This *process_id* represents the ID provided by the system and will determine whether this message can be open to public or not (*process_id* is assumed to be securely provided). In this case the tag of input message and output hash value will be either “High” or “Low” determined by *process_id*. When propagating tags in the inner sub-modules, all of the undefined tags will resolve to have the same dynamic tags, ensuring no false positives as shown in Table 2.

4.3 Designers’ Effort

SecChisel requires the system designers to add extra code to describe information flow policy and tags, as illustrated in Section 3. Table 2 Column “Chisel” shows the lines of code designer needs to write to implement AES RoCCs (including ones with HBTs) and SHA RoCC. Column “SecChisel” shows that tested modules require

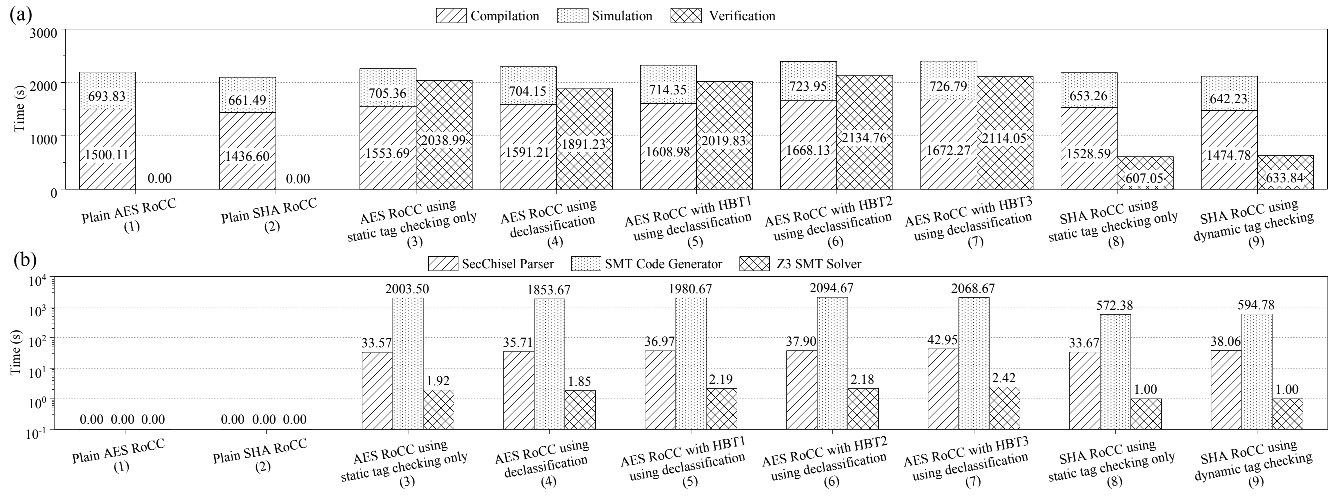


Figure 4: Evaluation of runtime of the SecChisel framework, times shown are averages of multiple runs. (a) Comparison of total compilation and simulation time vs. verification time for the different designs. (b) Runtime of SecChisel parser, SMT Code Generator, and Z3 SMT solver for the different designs.

Table 2: Effectiveness and designer effort in terms of lines of code of AES RoCC and SHA RoCC within Rocket Chip. "Formal Verification Result" shows effectiveness. FP represents False Positive verification result. "Chisel" column shows complexity of design in Chisel, without any security features. "SecChisel" column shows extra lines of code added to include information of security tags.

Module	SecChisel Features Used			Formal Verification Result	Chisel	Sec-Chisel
	Static Tag	Dynamic Tag	Declassi-fication			
AES RoCC v1 w/ static tags	✓	×	×	found FP	1097	+14
AES RoCC v2 w/ declassification	✓	×	✓	verified	1097	+17
AES RoCC v2 w/ HBT1	✓	×	✓	found HBT	1107	+17
AES RoCC v2 w/ HBT2	✓	×	✓	found HBT	1110	+17
AES RoCC v2 w/ HBT3	✓	×	✓	found HBT	1121	+17
SHA RoCC v1 w/ static tags	✓	×	×	found FP	1127	+25
SHA RoCC v2 w/ dynamic tags	×	✓	×	verified	1127	+28

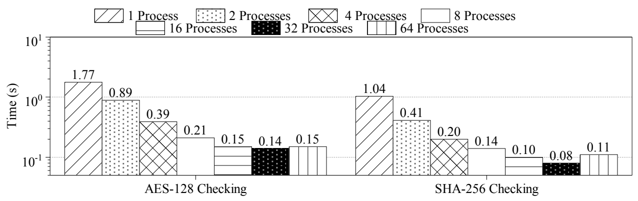


Figure 5: Runtime evaluation of effects of parallelizing the SMT code for different number of processors, on a server with 24 cores.

only tens of lines of code in order to do verification based on the original Chisel design. SecChisel implementation requires system designer to explicitly add security tags to critical variables, however, most of the variables in a module will have default (undefined) tags and the tags will be automatically resolved in SMT code generation step, and require no designer effort to specify such undefined tags.

4.4 Security Verification Performance

Figure 4 shows compilation plus simulation runtime and verification time as well as different parts of verification runtime for AES RoCC and SHA RoCC using different SecChisel features, and compares it with the runtime of unmodified Chisel tools for reference. Compilation includes all original Chisel flow before simulation shown in Figure 1. Verification consists of whole SecChisel verification flow circled by green dashed line in Figure 1. The verification runtime is in all cases slightly less than the compilation plus simulation runtime and the two can be done in parallel. The plain AES and SHA RoCC written in Chisel have no SecChisel features, so no related verification performance. Some small differences are due to the variation in performance of the server.

Shown in Figure 4a, SecChisel verification will not cause extra overhead to the original Chisel design. Verification can be hidden in normal compilation and simulation since the total verification time is always smaller than the compilation plus simulation time and the verification can be done in parallel with compilation and simulation. SecChisel's performance time overhead of changing RoCC to a secure module class is relatively small (AES RoCC's is around 3.8%, SHA RoCC's is around 2.5% by comparing Design 1, 2 with the other Designs in Figure 4a, respectively). As shown in Figure 4b, there is no significant performance time difference for the different cases. Finding the existence of an information leak by HBTs (Design 5, 6, 7 in Figure 4b) only introduced small time overhead comparing with Design 4 that does not have HBTs in it, for example.

4.5 Interference Table Evaluation

Interference table (IT) can be created during FIRRTL generation and will on average require 0.41s for AES RoCC and SHA RoCC, which is negligible compared with FIRRTL generation time shown in the SecChisel parser time in Figure 4b. SMT code generation with and without IT do not change much on the total runtime, but is offered a solution for use of trusted third-party modules for which source code is not available.

4.6 Z3 SMT Solver and Parallelization

Verifying the AES RoCC v2 and SHA RoCC v2 in Z3 SMT solver without parallelization cost 1.77s, 1.04s, respectively (Figure 5). For reference, AES RoCC generates 20832 lines of SMT code and SHA RoCC generates 9044 lines of SMT code, which is fully automatically generated based on the SecChisel code. Therefore, there is an approximately linear relationship between lines of SMT code and run time, including using different SecChisel features like dynamic tags, as shown in Figure 4b. Based on our experience with AES and SHA RoCC, we can estimate that one million of lines of Chisel code (far more than the current Rocket Chip code) will require around acceptable 2000s SMT runtime.

Parallelizing the SMT checks and running on multi-core system, as discussed in Section 3.6, can further reduce the run time of the SMT solver, as shown in Figure 5. For evaluation on a 24-core processor, the average improvement is about 20x.

4.7 Hardware Performance

The final design generated using the SecChisel code is identical to a design that would not contain any security tags or other SecChisel modifications. Specifically, SecChisel does not add any run-time components to the design as all the security verification is done at design-time. Therefore, the verified design will not add any performance overhead compared with the original design.

5 RELATED WORK

Both SecVerilog [23] and Caisson [14] use information flow tracking (IFT) to enforce security properties at compile time, but SecChisel is the first to be based on the Chisel language. Also, SecChisel is the first hardware security verification tool supporting nested modules without having to check individual module separately. And unlike SecVerilog, users of SecChisel need not to learn nor write SMT code. Meanwhile, Caisson does not support dynamic tags.

VeriCoq [5, 6], Formal-HDL [10], and ReWire [17] use formal proofs to verify security properties of a design by using the Coq language (VeriCoq, Formal-HDL) or the Haskell language (ReWire). Meanwhile, SecChisel uses approach based on SMT solver.

Orthogonal to compile-time verification, there are run-time security verification approaches including Sapper [13], GLIFT [21], or RTLIFT [2]. There are also projects which explore design verification after manufacturing, e.g., [22] [7], which are complimentary to our work.

6 CONCLUSION

We proposed SecChisel, the first hardware security verification framework based on Chisel. SecChisel was tested by implementing and verifying AES-128 and SHA-256 RoCC accelerators within a Rocket Chip RISC-V processor, We showed that SecChisel is able to detect information leaks due to hardware bugs or Trojans, and that SecChisel is fast and scalable when verifying designs such as the RISC-V Rocket Chip accelerators.

ACKNOWLEDGMENTS

This work was supported through NSF grant 1524680 and Semiconductor Research Corporation (SRC) award number 2015-TS-2633; and NSF grant 1813797 and SRC award number 2844.001.

REFERENCES

[1] AMD. 2016. AMD Memory Encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.
[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1691–1696.
[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
[4] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the Annual Design Automation Conference*. ACM, 1216–1225.
[5] Mohammad-Mahdi Bidmeshki, Xiaolong Guo, Raj Gautam Dutta, Yier Jin, and Yiorgos Makris. 2017. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP-Part II: Framework Automation. *IEEE Transactions on Information Forensics and Security* 12, 10 (2017), 2430–2443.
[6] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation. In *International Symposium on Circuits and Systems*. IEEE, 29–32.
[7] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. 2009. Hardware Trojan: Threats and emerging solutions. In *International High-Level Design Validation and Test Workshop*. IEEE, 166–171.
[8] David Champagne and Ruby B Lee. 2010. Scalable architectural support for trusted software. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.
[9] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[10] Yier Jin and Yiorgos Makris. 2013. A proof-carrying based framework for trusted microprocessor IP. In *International Conference on Computer-Aided Design*. IEEE, 824–829.
[11] Ruby B. Lee, Peter Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 2–13.
[12] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. 2016. *Specification for the FIRRTL Language*. Technical Report UCB/EECS-2016-9. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html
[13] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 97–112.
[14] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 109–120.
[15] David Lie, John C. Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. 2003. Specifying and verifying hardware for tamper-resistant software. In *Proceedings of Symposium on Security and Privacy*. IEEE, 166 – 177.
[16] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*.
[17] Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard Allwein. 2015. Semantics Driven Hardware Design, Implementation, and Verification with ReWire. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*. ACM, Article 13, 10 pages.
[18] RISC-V 2019. RISC-V Foundation. https://riscv.org/.
[19] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the International Conference on Supercomputing*. ACM, 160–171.
[20] Jakub Szefer and Ruby B Lee. 2012. Architectural support for hypervisor-secure virtualization. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 437–450.
[21] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, Vol. 44. ACM, 109–120.
[22] Riad S Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. 2016. Verifiable asics. In *Proceedings of Symposium on Security and Privacy*. IEEE, 759–778.
[23] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.