Tutorial: SecVerilog

Rui Xu **Danfeng Zhang**

July 6, 2015

SecVerilog [3] is a new hardware description language (HDL) with fine-grained information flow control. SecVerilog extends Verilog with annotations that support comprehensive, precise reasoning about information flows, including flows via timing, at compile time. The benefit is that hardware designs can be verified mostly as-is, with little run-time overhead.

SecVerilog fits into a typical design flow very well, as demonstrated in Figure 1. As input, SecVerilog takes Verilog code with security labels (e.g., `{D1}`, `{D2}` and `{L}`). These labels specify the information flow policy to be verified on a hardware design. Once the verification succeeds, security labels are removed, resulting in standard Verilog code. Such Verilog code is then used in a typical hardware design flow. The verified hardware provably obeys the information flow policy specified in the original SecVerilog code, guaranteed by the type system of SecVerilog.

A distinguishing feature of SecVerilog is the support of dependent labels. Dependent labels allow hardware resources to be shared across security domains. In the two-input mux example shown in Figure 1, the security domain of `out` depends on the run-time value of `sel`: the domain is `D1` when `sel` is 0, and `D2` when `sel` is 1. Such restrictions can be concisely specified by a dependent label `{Domain sel}`, where `Domain` is a function from values to labels (defined in a separate file).

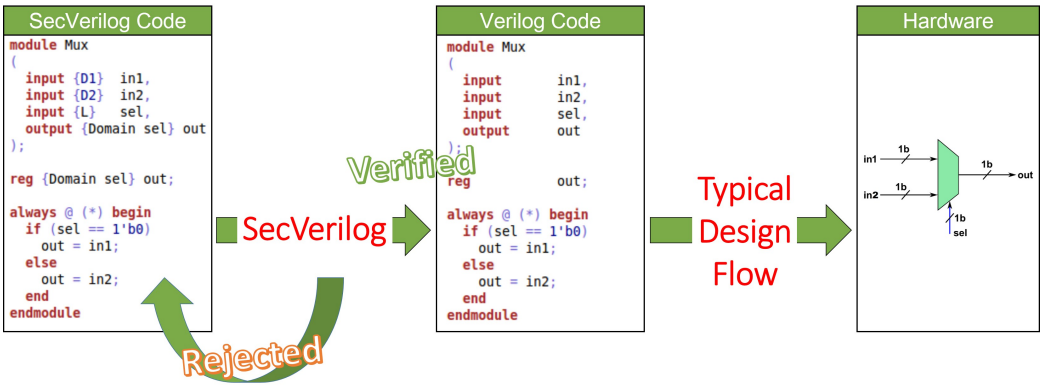


Figure 1: SecVerilog overview.

This tutorial assumes basic knowledge of using Verilog for hardware design. The focus of this tutorial is on how to use SecVerilog to verify information-flow security of hardware designs.

The SecVerilog implementation is based on the open source tools `Icarus Verilog (Iverilog)` [1] (for compiling Verilog modules), and Z3 [2] (for verification). The source code of Iverilog is included in the installation package of SecVerilog, while you have to install Z3 separately.

1 How to install SecVerilog

You need to download the installation package, and uncompress the package. The installation package contains two folders:

- `SecVerilog`: the SecVerilog compiler.
- `Examples`: SecVerilog code examples.

In this tutorial, we refer to the first folder as `HOME` and the second as `EXAMPLES`. To build and install SecVerilog, execute the following commands:

```
cd HOME
./configure
make
make install
```

If you want SecVerilog being installed in a specified directory, say `DIR`, use the following commands instead:

```
cd HOME
./configure --prefix = DIR
make
make install
```

If all steps work well, SecVerilog is successfully installed into the folder specified by the `DIR` parameter, or a default folder (e.g., /usr/local/bin in Linux) when `prefix` is absent. The installation may require other packages, such as Gperf. Please install them when necessary.

2 How to use SecVerilog

We use the `oneway` example under the `EXAMPLES` folder to explain the usage of SecVerilog. There are two ways of using SecVerilog: manually invoking `iverilog` and `z3` binaries (Section 2.1), or automate the verification with a Perl script (Section 2.2).

```
; Domain(0)=L1, Domain(1)=L2
(declare-fun Domain (Int) Label)
(assert (= (Domain 0) L1))
(assert (= (Domain 1) L2))
```

```
; new lattice elements L1,L2
(declare-fun L1 () Label)
(declare-fun L2 () Label)

; lattice structure
(assert (leq L1 L2))
```

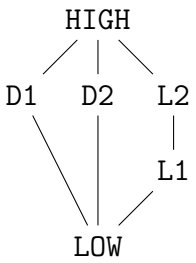
Figure 2: Declaration of type-level function (left) and lattice (right).

2.1 Usage 1: `iverilog` and z3 binaries

The `iverilog` binary takes Verilog files with security labels as input, and produces a Z3 file as output. Besides the parameters of Icarus Verilog, the `iverilog` command takes three new parameters:

- `-F depfunfile`: provide declarations of type-level functions. For example, the left of Figure 2 shows `EXAMPLES/oneway.fun`. This file defines a type-level function `Domain`, where `Domain(0)=L1` and `Domain(1)=L2`. Security levels `L1` and `L2` are declared by the `-l` option as below.

- `-l latticefile`: provide an extension to SecVerilog's default security policy (in the form of a lattice of security levels). By default, SecVerilog supports a "diamond" lattice with four levels: `LOW`, `D1`, `D2` and `HIGH`. With the extension on the right of Figure 2, the new security policy forbids information from flowing "down" in the lattice below:

  Note that by default, the new added labels are always lower than `High` and higher than `Low`.

- `-z`: verification only. That is, SecVerilog quits after type-checking.

To verify program `oneway.v` in the folder `EXAMPLES`, run the following command in the same folder:

```
iverilog -F oneway.fun -l oneway.lattice -z oneway.v
```

The expected result is a Z3 file `oneway.z3` in the same folder. Assume Z3 is already installed, the following command completes the verification of file `oneway.v`:

```
z3 -smt2 oneway.z3
```

For this example, the expected output is:

```
unsat
unsat
unsat
sat
```

Here, `unsat` (`sat`) means the SecVerilog statement generating the corresponding constraint is secure (insecure). Here, the last constraint is generated by the last assignment in `oneway.v`, which is `d1 = d2`. Since the labels of `d1` and `d2` are `L1` and `L2` respectively, an insecure flow from `L2` and `L1` is correctly identified by SecVerilog in this example.

2.2 Usage 2: the `secverilog` script

To ease the use of SecVerilog, the installation package contains a Perl script template `secverilog` under the folder `EXAMPLES`. The script completely replaces calling `iverilog` and `z3` manually, as described in the previous section. Moreover, the script automatically maps security violations (satisfiable constraints) back in to the Verilog code being verified. This is useful for fixing security bugs in hardware designs.

The script can be used as is if both `iverilog` and `z3` are under the PATH environment variable. Otherwise, modify `$z3home` and `$iveriloghome` accordingly to the binary folders of `z3` and `iverilog` respectively.

Once the template is configured properly and renamed as `secverilog`, we can verify `oneway.v` by one command:

```
./secverilog -F oneway.fun -l oneway.lattice -z oneway.v
```

or equivalently,

```
perl secverilog -F oneway.fun -l oneway.lattice -z oneway.v
```

The expected output is:

```
Compiling file oneway.v
Verifying file oneway.v
fail
(assert  (not(leq L2  L1)))    ; d1 = d2 @oneway.v:38
Total: 1 assertions failed
```

Here, the script clearly points to a security violation at line 38 of `oneway.v`.

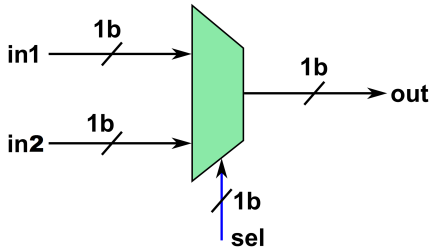


Figure 3: Block diagram for a two-input mux: a mux module with two one-bit input ports, a one-bit output port, and a one-bit select control input.

3 Design secure hardware with SecVerilog

In this section, we create our very first Verilog hardware module and learn how to verify this module with SecVerilog. We use the two-input mux hardware module shown in Figure 3 as a running example. We start from building a RTL behavioral model, and then use SecVerilog to verify its security.

3.1 RTL behavioral model of a two-input mux

Figure 4 shows the Verilog code which corresponds to the diagram in Figure 3. Lines 3-6 declare all parameters used in the `Mux` module. `in1` and `in2` represent two input ports. The `sel` variable controls which input signal can be passed to the output port. Because this mux only has two input ports, `sel` is a one-bit signal. The output port is named `out`, which is also one-bit. Lines 11-16 model the internal behavior of the module. When `sel` is zero, `in1` is passed to the output port. Otherwise, the output port will select the other input signal, `in2`.

3.2 Define a security policy

To verify the security of a hardware design, we first need to define the information-flow policy that the design should obey. Such policy is specified as a lattice of security levels. A lattice of security levels restricts information flow: information with lower security levels can flow to information with higher security levels, but the opposite direction is prohibited.

Once a security lattice is defined, and information (variables) in the hardware design is properly annotated with security labels, SecVerilog can, at compile time, verify if the aforementioned information-flow policy is enforced.

```
1  module Two_Input_Mux
2  (
3      input in1,
4      input in2,
5      input sel,
6      output out
7  );
8
9  reg out;
10
11 always @ (*) begin
12     if (sel == 1'b0)
13         out = in1;
14     else
15         out = in2;
16 end
17 endmodule
```

Figure 4: Two-Input mux: RTL behavioral code corresponding to the diagram in Figure 3.

In order to specify the correct lattice structure for the module to be verified, we need to figure out the security levels, and the relationship between them. Returning to our running example, we assume that there are three different security levels, namely `D1` (domain 1), `D2` (domain 2), and `LOW` (public) in this mux module. Any information flow between `D1` and `D2` is not allowed. As the name of indicates, `LOW` is the lowest secure level: information from this level can be passed to any security levels including itself. However, the opposite direction is prohibited.

The proper lattice is defined in a lattice file. For this example, the security lattice is part of the default lattice in SecVerilog already. So no lattice file is needed. The default lattice, as depicted in Figure 5, is declared by the code in the same figure. Lattice declaration has two parts — lattice elements (security levels) and lattice structure. Here, lines 2–5 declare four security levels: `LOW`, `HIGH`, `D1` and `D2`. The relation among these levels are specified by lines 8 and 9. Line 8 specifies that `LOW` is the lowest (least restrictive) level among all levels, meaning that information with `LOW` can flow to all levels. The dual of `LOW` is `HIGH`, where all information can flow into, as specified in line 9. Since there is no `leq` relation on levels `D1` and `D2`, any flow between these two levels is disallowed.

```
1 ; lattice elements
2 (declare-fun LOW () Label)
3 (declare-fun HIGH () Label)
4 (declare-fun D1 () Label)
5 (declare-fun D2 () Label)
6
7 ; lattice structure
8 (assert (forall ((x label)) (leq LOW x)))
9 (assert (forall ((x label)) (leq x HIGH)))
```

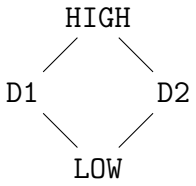


Figure 5: Lattice structure for the two-input mux module.

3.3 Verifying the two-input mux module

In this example, the security concern is that inputs from different security levels are fed to corresponding domains, with a shared output signal. The module needs to avoid inputs from `D1` flowing to `D2`, and vise versa. In another word, at each clock cycle, the input signal's level should be consistent with that on the output port. So the following restrictions should be obeyed:

- Input `in1` belongs to `D1`
- Input `in2` belongs to `D2`
- Ouput `out`'s domain depends on the value of the signal `sel`. If the `sel` is zero, `out` belongs to `D1`. Otherwise, it belongs to `D2`

Figure 6 presents the SecVerilog code for the two-input mux (available as `EXAMPLES/mux.v`). Compared to Figure 4, the baseline Verilog code, the only difference is the security labels for variable declarations. SecVerilog uses braces to denote security labels. In the simplest case, a security label is simply a security level in the security lattice. For example, at line 3, {`D1`} indicates that the input signal `in1` belongs to `D1`.

More interesting is the label of `out`, which depends on the run-time value of `sel`. At line 6, `Domain` represents a type-level function, which maps 0 to `D1` and 1 to `D2`. This type-level function is declared in a separate function file, `EXAMPLES/diamond.fun`, as shown in Figure 7. Here, the declaration is very straightforward: the first line declares a function named "Domain", and the rest two lines specifies the desired mapping.

In general, the security labels in SecVerilog can be an application of type-level functions, such as the label {`Domain sel`} for `out`. This label exactly captures the restrictions on `out`'s value: when `sel` is zero, `out` belongs to `D1`; when `sel` is one, `out` belongs to `D2`.

```
1  module Two_Input_Mux
2  (
3      input {D1}           in1,
4      input {D2}           in2,
5      input {L}            sel,
6      output {Domain sel} out
7  );
8
9  reg {Domain sel} out;
10
11 always @ (*) begin
12     if (sel == 1'b0)
13         out = in1;
14     else
15         out = in2;
16 end
17 endmodule
```

Figure 6: SecVerilog code of the two-input mux example: SecVerilog version code for the two-input mux module in Figure 4.

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

Figure 7: Type-Level function `Domain` for the two-input mux example.

The next step is to compile the above code with SecVerilog, and use the `Z3 solver` to verify whether is there any violation of the security policy. As described in Section 2, this can be done as follows:

```
iverilog -F diamond.fun -z mux.v
z3 -smt2 mux.z3
```

The expected output is a series of `unsat`, meaning that there is no security violation in the SecVerilog code being analyzed. For better debugging support, we recommend the `secverilog` script for verification. As described in Section 2, we can simply run the following command to verify `mux.v`:

```
perl secverilog -F diamond.fun -z mux.v
```

The expected output is:

```
Compiling file mux.v
Verifying file mux.v
verified
Total: 0 assertions failed
```

4 Advanced use of SecVerilog

The two-input mux example illustrates the basic usage of SecVerilog. This part discusses some advanced aspects of SecVerilog, which may be useful for advanced users.

4.1 User-Defined lattice and type-level functions

SecVerilog supports arbitrary user-defined lattice structures and type-level functions. They are defined in separate files and fed to SecVerilog by parameters `-l` and `-F` respectively.

By default, SecVerilog supports a "diamond" lattice with four labels: `LOW`, `D1`, `D2` and `HIGH`. With extensions, such as the right of Figure 2, new lattice elements (security levels) can be added in straightforward ways.

By default, SecVerilog supports a type-level function `LH`, which maps 0 to `LOW`, and 1 to `HIGH`. With extensions, such as the left of Figure 2, new functions can be added in straightforward ways as well.

4.2 Rewrite hardware designs for SecVerilog

In the ideal case, designers only need to add security labels for variable declarations, if the hardware design being verified is secure. However, due to

the imprecision of SecVerilog's type system[1], secure designs sometimes need to be slightly modified to overcome such imprecision.

One way to overcome such limitation is to add extra conditions to rule out impossible states. Next, we illustrate such limitation by examples.

4.2.1 Reasoning about impossible states

For security, SecVerilog must conservatively consider all possible hardware states that might occur at run time. However, sometimes, the hardware designers write Verilog code in a way that gets the design away from insecure circumstances. In general, it is hard for SecVerilog to reason about the impossibility of such insecure states at compile time. We use the example in Figure 8 to illustrate this situation.

The left of Figure 8 shows a secure design, which is (incorrectly) rejected by SecVerilog. In this example, the module selectively passes the input request `req` to `req_low` with label `LOW`, or `req_d1` with label `Domain1`, or `req_d2` with label `Domain2`, according to the value of `sel`. Here, the type-level function `Domain` maps 0 to `L`, 1 to `D1`, and 2 to `D2`.

This design is secure because the signal `sel` only holds three possible values (0, 1, 2). This is enforced in other pieces of code. In another word, in the last branch of the source code, the security level of `req` must be `D2`. Therefore, there is no security violation in this program. However, since SecVerilog needs to consider all possibilities, including the impossible case where `sel` is 3, this program is rejected.

The right of Figure 8 shows a functionally equivalent hardware design, which can be verified by SecVerilog. The only added code is the branch condition for the last branch. Such added condition helps SecVerilog to rule out the impossible insecure state, so that the design passes the SecVerilog verification.

4.2.2 Reasoning about invariants

Another source of imprecision is the failure of SecVerilog to reason about invariants established in hardware designs. We use the example in Figure 9 to illustrate this case.

In this example, the type-level function maps 0 to `D1` and 1 to `D2`. The code on the left seems insecure, because when `sel` is 1, the assignment to `do_enq` effectively leaks information with label `D2` (the value of `enq_rdy`) to the security domain `D1` (`do_enq`). However, this design is secure because the

[1]Such imprecision mostly comes from the program analyses adopted in the current implementation of SecVerilog.

```
input [1:0] {L}  sel;
input [3:0] {L}  req_low;
input [3:0] {D1} req_d1;
input [3:0] {D2} req_d2;
//Domain 0=L, 1=D1, 2=D2
output[3:0] {Domain sel} req;
if (sel == 0)
    req_low = req;
else if (sel == 1)
    req_d1 = req;
else
    req_d2 = req;
```

```
input [1:0] {L}  sel;
input [3:0] {L}  req_low;
input [3:0] {D1} req_d1;
input [3:0] {D2} req_d2;
//Domain 0=L, 1=D1, 2=D2
output[3:0] {Domain sel} req;
if (sel == 0)
    req_low = req;
else if (sel == 1)
    req_d1 = req;
else if (sel == 2)
    req_d2 = req;
```

Figure 8: Rewriting hardware designs to aid SecVerilog.

```
input  {D1} enq_val;
//Domain 0=D1, 1=D2
input  {Domain sel} enq_rdy;
input  {L}  sel;
output {D1} do_enq;
assign do_enq = enq_val
              & enq_rdy;
```

```
input  {D1} enq_val;
//Domain 0=D1, 1=D2
input  {Domain sel} enq_rdy;
input  {L}  sel;
output {D1} do_enq;
if (sel == 1'b0)
    do_enq = enq_val
           & enq_rdy;
else
    do_enq = 1'b0;
```

Figure 9: Change of interface leads to added code.

hardware designer restricts `enq_val` to be zero whenever `sel` is 1. Therefore, the value of `do_enq` is always zero whenever `sel` is 1—no information is leaked from `D2` to `D1`.

To aid SecVerilog to observe the relation of `sel` and `enq_val`, we can rewrite the code on the left as the one shown on the right of Figure 9. The aforementioned invariant is now explicit in the rewritten code, so that SecVerilog correctly accepts this program.

5 Limitations in current implementation

The current implementation of SecVerilog has the following limitations.

- SecVerilog does not allow adding labels to the `parameter` data type. One workaround is to turn these types into `input` ports with constant values.

- When sub-modules are in the same file with a main module, these sub-modules are not verified by SecVerilog. One workaround is to put sub-modules in separate files.
- For dynamic labels, inputs for a mapping function must be a complete variable. SecVerilog does not support a dependency on certain bits of a variable yet. For example, a label `{Domain x[0]}` is not allowed.
- Each bit in a variable must have the same label.
- SecVerilog currently does not support `task` and `function`. Such code blocks will be ignored during verification. Designers can inline these blocks, or turn these blocks into `module`s to avoid this limitation.
- Although `if (x == 1'b1)` is equivalent to `if (x)` when `x` is a boolean, SecVerilog requires the former to establish an invariant that `x` is 1.

References

[1] Icarus Verilog. http://iverilog.icarus.com/.

[2] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[3] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proc. 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.