ASSURE
User Guide

Christian Pilato[1]

July 31, 2020

[1]Politecnico di Milano, Italy (christian.pilato@polimi.it)

Contents

Chapter 1
Preface

This document describes ASSURE, a prototype CAD tool[1] for applying **logic locking** at the *register-transfer level* to raise the semantics of circuit obfuscation, while keeping the designs compatible with current industrial design flows. At RTL, the IP microarchitecture is defined (so it can be applied to pre-existing components), while part of the semantic information (e.g., constant values) is present in the circuit design, enabling effective protection (see Fig. 1.1). The approach does not require any modification to EDA tools. In collaboration with BOEING, we evaluated ASSURE RTL obfuscation tool developed by Politecnico di Milano and New York University on a set of pre-existing IP cores described in Verilog. The resulting obfuscated circuits have been validated and evaluated without modifying the EDA flow used for the original designs.

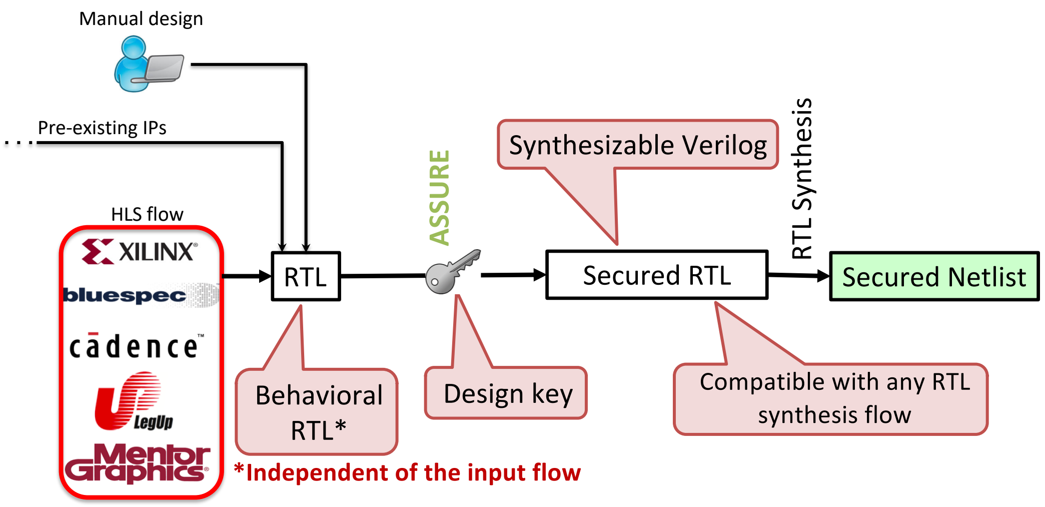

Figure 1.1: ASSURE design flow: Operating at RTL allows designers to protect more semantic information.

1.1 Integrated Circuit (IC) Design Flow

The traditional design flow of a hardware IP core is shown in Fig. 1.2. The synthesis flow starts from a specification of the circuit functionality in a *hardware description language* (HDL). At this point, the circuit is specified at the register-transfer level (RTL) and this description can be obtained with several methods – manual design, high-level synthesis and hardware generators. Many chip designs are created by composing IP modules created using any of these methods. So, it is extremely important to have an obfuscation method that covers all of these cases right after the integration step and before the actual logic synthesis.

[1]The development of ASSURE is a joint collaboration between Politecnico di Milano and New York University.

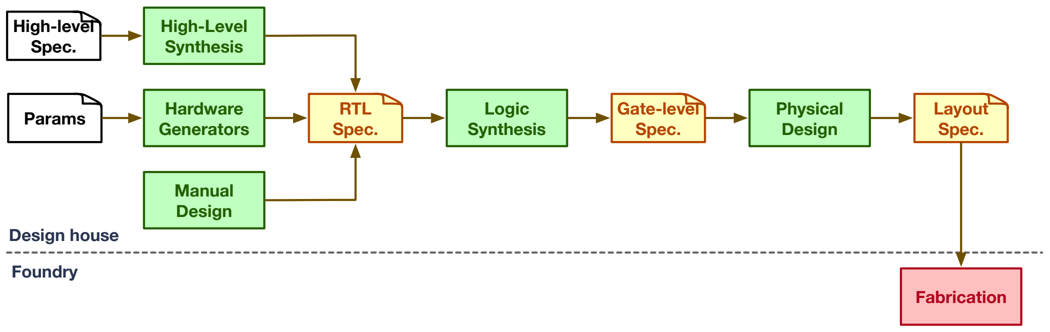

Figure 1.2: IC design flow with third-party (untrusted) foundry.

Given a technology library (i.e., a description of the available gates in the target technology) and a set of constraints, logic synthesis is performed on the RTL description to obtain a gate-level netlist. Several optimizations are applied at this stage to reduce area and improve timing, including constant propagation, state encoding, memory inference and optimization. These optimizations can embed sensitive information into the netlist that cannot be further protected. For example when a multiplication by 2 or powers of 2 is replaced with shift operator, the attacker can glean information about the constant (i.e., power of two value) either directly from the optimized design or indirectly from the side channels such as power and timing. Physical design is then performed to obtain the GDSII format layout files, which are then given to the foundry for fabricating the chip.

While RTL descriptions are hard to match with high-level specifications, they are used as a golden reference in the rest of the flow to verify that no errors are introduced by our obfuscation tool using formal equivalence checker methods.

1.2 Threat Model: An Untrusted Foundry

The main goal of the untrusted foundry or a rogue in the untrusted foundry is to identify the type and sequence of operations of the IC in order to replicate its functionality. We assume that the adversary can access the GDSII layout files of the obfuscated design. From this, the attacker can reverse engineer the *gate-level netlist* and, in turn, the types of RTL modules used in the design and their connections. The attacker can perform simulations with different inputs and different key values as well as re-syntheses with different key values. However, we assume that the attacker has **no access to an activated circuit, also called an ORACLE**, so SAT attacks cannot be applied. This is a reasonable assumption for low-volume ICs that are not going to the market (e.g., DoD circuits) or when the IC is produced in batches and the foundry is the first place where the design files are outsourced. We also assume that the attacker has no prior information on the design. In this scenario, security is guaranteed when all input keys are equally plausible so that the attacker can only make random guesses without knowing whether a key is correct or not. The protection is intended to reveal no more information what is already available before obfuscation. This technique can be used for “IC metering” to thwart over production of the design by the foundry. For example, if a chip design company is developing a design for companies A and B. The chip design company can use different keys for companies A and B even though the design is the same. If the foundry over produces chips from company A and sells it to company B, it will not work since keys from company B cannot unlock this design.

1.3 Summary

The rest of this document is organized as follows:

- Chapter 2 describes the methodology implemented by ASSURE;
- Chapter 3 presents the tool, including information for installation and configuration;
- Chapter 4 describes the interface of the locked designs generated by ASSURE.
- Chapter 5 describes how to perform formal verification on the locked designs generated by ASSURE.

Chapter 2

RTL Logic Locking with ASSURE

ASSURE is a command-line framework that converts Verilog RTL into secured Verilog RTL to hide the **essential semantics** of the given IP core. We identify the following minimal set of semantic elements necessary to replicate an IP functionality:

- *constant values* may contain sensitive information (e.g., filter coefficients).
- *operations* determine circuit functionality.
- *control-flow* describes the execution flow (i.e., which operations are executed under certain conditions).

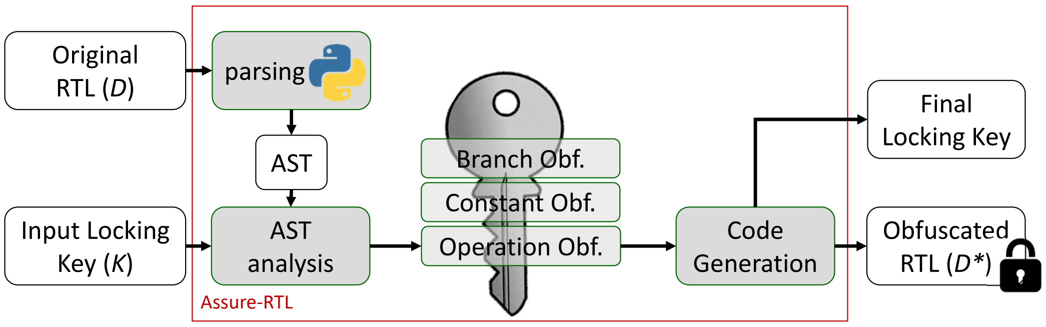

Figure 2.1: Overall structure of the ASSURE framework.

The structure of ASSURE is shown in Fig. 2.1. Given any RTL design D and an extra key parameter K (unknown to the foundry), ASSURE generates a design D^* that has the same behavior (i.e., it produces the same results) when the key value K is provided. This equivalence is formally verified. Functionality of D^* is harder to understand without having the parameter K. The essential idea is that different K values correspond to the resulting circuit having different semantics, especially when the attacker has no additional information. In short, these variants are *indistinguishable* from one another.

ASSURE starts by parsing the input Verilog description and creating the corresponding *abstract syntax tree* (AST). The next phase (*AST analysis*) analyzes the design AST with a depth-first search to identify the semantic elements to protect and applying the corresponding transformations (see Section 2.1). The framework offers several options for the designer to control how to apply obfuscation: it is possible to select on which modules to apply obfuscation, a “budget” in terms of key bits for each technique, or to exclude part of the design from obfuscation. While the number of available key bits for the parameter K is a design constraints as it depends on the technology for storing them in the final circuit, the number of bits needed to obfuscate the semantic elements is application-dependent as it is related to the complexity of the algorithm to protect. When ASSURE needs more key bits than the ones available in the parameter K, the designer can decide to terminate the analysis when reaching the number of available bits or to reuse the input key bits to obfuscate more elements. The output RTL design as the same external interface as the original module, except an additional input port that must be connected to the parameter K after fabrication. In addition, it also generates the final locking key that combines the input values that have been used for obfuscation and the values extracted by the ASSURE flow from the circuit. The security level of each design is then evaluated in terms of number of obfuscated elements (constant, operations, and branches) and number of bits effectively used for obfuscation. Detailed reports thus are generated by ASSURE. Scripts for formal verification between the original and the obfuscated designs are also automatically generated. The key value used for formal verification can be also changed to verify that alternative keys cannot activate the chip, i.e., they produce a design that is equivalent to the original one.

The log from the ASSURE tool provides information to the designer concerning the number of bits available for each of the obfuscation techniques and at what module in the design hierarchy. This will aid the designer in choosing what module to obfuscate and how many bits to budget for a given obfuscation when they are working with an overall budget.

ASSURE operates directly on the IP function rather than on its gate-level implementations. Since it is implemented directly at the RTL, it does not require modifications to the tools already used in the different design phases.

2.1 ASSURE: Three Obfuscation Techniques

In this section, we present the techniques used to thwart the reverse engineering of an IP core. Each technique targets an essential element to protect by using a specific part of the logic key K.

2.1.1 Constant Obfuscation

We remove selected constants from the design and use their values as locking key bits. A constant $CONST$ in the given RTL replaced by a portion of the locking key, as shown in Fig. 2.2. For example, consider the RTL operation `b = a + 5'b01010` to protect. We replace the 5-bit constant with a key signal K_{CONST} `= 5'b01010` and rewrite the RTL as `b = a +` K_{CONST}. The original functionality is preserved only when the correct value is loaded into K_{CONST}.

Constant obfuscation may prevent subsequent logic optimizations (e.g., constant propagation and trimming). Also, constant obfuscation is not applied to values that are not significant to obfuscate (e.g., reset values) or when it can change how the logic synthesis tool infers the semantics of the design (e.g., reset polarity).

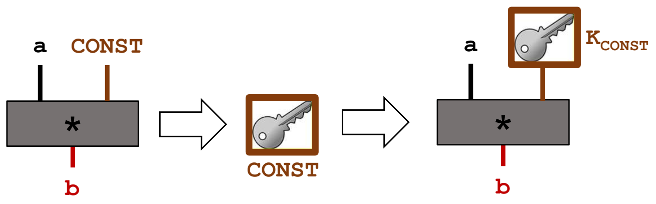

Figure 2.2: Example of constant obfuscation.

2.1.2 Operation Obfuscation

To obfuscate arithmetic RTL operations, we create dummy operations that are dynamically selected based on the value of the key bits. The additional operators are connected to the same inputs of the original operation (so no additional multiplexers are needed) and the outputs of the original and dummy operations are multiplexed based on the assigned key bits, as shown in Fig. 2.3. While this technique does not introduce any functional change in the computation, critical path delays may be affected due to the insertion of the dummy operation and the additional multiplexer.

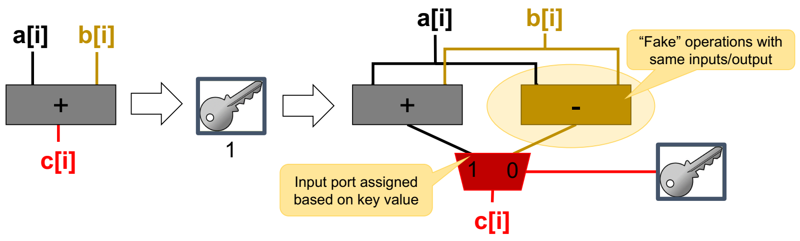
a[i]
b[i]
+
c[i]
1
a[i]
b[i]
+
-
"Fake" operations with same inputs/output
Input port assigned based on key value
1
0
c[i]

Figure 2.3: Example of operation obfuscation.

2.1.3 Control-Flow Obfuscation

Each control condition in the behavioral RTL is obfuscated with a key bit as follows: $(\texttt{condition}) \wedge K_j$. Also, to maintain semantic equivalence, the two subsequent branches are reordered in order to reproduce the correct control flow when $K_j = 1$ because the XOR obfuscation gate inverts the value of the condition, as shown in Fig. 2.4.

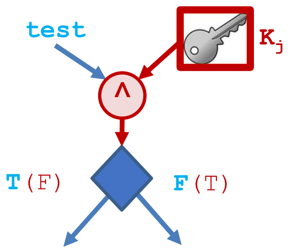

Figure 2.4: Example of branch obfuscation.

The same technique can be applied also to conditional expressions (e.g., multiplexers) to create alternative data flows. Indeed, with incorrect keys, the operation will execute with different values. The attacker cannot determine the correct control flow or the correct operation inputs without having the correct key bits. Similar to constant obfuscation, control-condition obfuscation is not applied when it is not relevant. For example, reset and update processes are not obfuscated.

Chapter 3

Tool Overview

ASSURE is a Python-based command-line tool that generates the Verilog code for an input design based on a set of obfuscation options. This chapter describes the tool interface and how to configure and execute the tool. The list of ASSURE options are as follows:

```
Usage: assure.py [options]

Options:
  -h, --help            Show this help message and exit
  -V, --version         Show the version
  -v, --verbose         Verbose execution
  -I INCLUDE, --include=INCLUDE
                        Include path
  -D DEFINE             Macro Definition
  -f FILE_LIST, --file-list=FILE_LIST
                        Specify file with command-line options for the Verilog
                        compiler
  -t TOP, --top=TOP     Specify the name of the module to obfuscate
  --locking-key=INPUT_KEY
                        Specify the input locking_key file
  -o OUTPUT_DIR, --output=OUTPUT_DIR
                        Specify the name of the output directory
  --obfuscate-const=OBFUSCATE_CONST
                        Specify the minimum number of bits for the constants
                        to obfuscate
  --obfuscate-branch    Obfuscate conditional branches
  --obfuscate-ops       Obfuscate operators
  --obfuscate           Obfuscate entire design
  --enable-key-reuse    Reuse key bits (if needed)
  --key-budget=KEY_BUDGET
                        Specify the key budget (const,op,branch)
  --modules=MODULE      Specify the list of modules to obfuscate
```

Specifically, the general options are:

- `help` prints the above message window.
- `version` prints the tool version.
- `verbose` increases the verbosity of the output messages. Increasing the verbosity level allows the user to get additional information about the tool execution.

There are two modes to specify the input design:

- by listing the input files as tool arguments and adding specific compilation options:
  - `include` specifies the directories where included files are located.
  - `define` specifies the definition of pre-processing macros for the design.
- by specifying a compilation file with the option `file-list`.

In both cases, the designer needs to specify the top module (option `top`) from which ASSURE will start the obfuscation.

The options for controlling the obfuscation are:

- `locking-key` specifies the input key to be used for obfuscation
- `obfuscate-const` enables constant obfuscation. The additional parameter specifies the minimum bitwidth for a constant to be obfuscated. If the parameter is `0`, all constants are considered.
- `obfuscate-ops` enables operation obfuscation.
- `obfuscate-branch` enables control-flow obfuscation.
- `obfuscate` is a short option that enables all the three techniques mentioned above (constants, operations, and control-flow). It is equivalent to the following list of options:
  `-obfuscate-const 0 -obfuscate-ops -obfuscate-branch`.
- `enable-key-reuse` enables the reuse of the input key bits to obfuscate the entire design. It is equivalent to providing a larger key with the input key bits repeated as many times as needed to obfuscate all elements of the design.
- `key-budget` specifies a key-bit budget for each technique. The parameter is a comma-separated string with the values for the three techniques, in the following order: constants, operations, control-flow. When no constraint is imposed for a technique, the designer can use the symbol "*".
- `modules` specifies a text file that contains the list of modules to obfuscate. If provided, only the listed modules are considered for obfuscation. Otherwise, all modules are taken into account.

The output options are:

- `output-dir` specifies the output directory where all results will be stored.

3.1 Configuration, Installation, and Execution

ASSURE has been implemented in Python and tested under Linux Ubuntu 18.04 LTS and MacOS 10.15 and CentOS 7. MNEMOSYNE requires the following libraries for compilation and execution:

- python3 (ver. >= 3.6)
- icarus-verilog (ver. >= 10.1)

Once these dependencies are satisfied, the tool can be deployed in any directory. The tool does not require any further installation and is ready to use as is.

3.1.1 Execution Tutorial

To execute Assure on a pre-existing design, it is necessary to complete the following step:

1. setup the environment to execute the tool (optional);
2. prepare the input design;
3. prepare the input key;
4. select the output directory;
5. select the obfuscation option;
6. run the tool.

Environment Setup. The first step is to setup the environment to execute the tool as follows:

```
$> export ASSURE_SRC=<path_to_assure_src>
```

This step is optional. If performed, the tool can be then invoked as follows:

```
$> python3 ${ASSURE_SRC}/assure.py [options]
```

Otherwise, the designer needs to specify the path to reach the file `assure.py` contained in the directory `src` of the tool distribution. For example, let us assume that the tool is executed in the directory `tests/i2c_slave/tgt`, the tool can be invoked as follows:

```
$> python3 ../../../src/assure.py [options]
```

Input Design Specification. The next step is to list the input Verilog files, the compilation options for running the tool on the input design, and the name of the top module. For example, let us assume we have a directory `tests/i2c_slave` that contains a directory `src` where the input files are stored and a directory `tgt` where we want to run the tool. The following files are stored in the directory `src`:

Content of directory src

```
i2cSlave.v
i2cSlaveTop.v
registerInterface.v
serialInterface.v
i2cSlave_define.v
timescale.v
```

However only the first four files are actual design files, while the last two are files included in the code with the Verilog `‘include` directory. So, it is necessary to specify the include path where these files are stored. If the files are listed as arguments, the tool can be invoked as follows from the directory `tgt`:

```
$> python3 ${ASSURE_SRC}/assure.py ../src/i2cSlave.v ../src/i2cSlaveTop.v \
                                   ../src/registerInterface.v ../src/serialInterface.v \
                                   -I../src --top i2cSlaveTop
```

where the option `-I` is used to specify the include directory and the option `--top` for the name of the top module. As an alternative, the list of Verilog files and include options can be listed in a file that has the format of the command-line operations for the commercial simulators. So, the same options can be written in the following file:

File: file_list.f

```
+incdir+../src
../src/i2cSlaveTop.v
../src/i2cSlave.v
../src/registerInterface.v
../src/serialInterface.v
```

In this case, the tool can be invoked as follows:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop
```

⚠ ASSURE leverages Icarus Verilog as a front-end to parse the input Verilog code. So, it has the same limitations as the Icarus project.

Input Key Preparation. The next step is the preparation of the input locking key. ASSURE offers the possibility of specifying a user-defined key file or automatically generating one. In the former case, it is necessary to create a text file, where the key bits are specified using the Verilog constant format. So, a 16-bit constant can be specified as follows:

File: input.key

```
16'b0101110101010100
```

or:

File: input.key

```
1101110101010100
```

or:

File: input.key

```
1101_1101_0101_0100
```

or:

File: input.key

```
16'hDD54
```

⚠ ASSURE assumes that the key bits are specified from the most-significant bit (MSB) to the less-significant bit (LSB). Lines starting with the symbol # are considered as comments and ignored.

It is also possible to automatically generate a locking key as follows:

```
$> python3 ${ASSURE_SRC}/generate_key.py -b <num_bits> -o <file_name>
```

So, for example, a 16-bit key can be generated as follows and saved in the file `new_input.key`

```
$> python3 ${ASSURE_SRC}/generate_key.py -b 16 -o new_input.key
```

In all cases, once the locking file has been determined, the tool can be invoked as follows:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop \
                                   --locking-key ./input.key
```

Output Directory Selection. The default output directory is a subdirectory, named `work`, of the current directory. It is possible to specify a different name or a different target path as follows:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop \
                                --locking-key ./input.key -o <new_target_dir>
```

⚠ ASSURE assumes that all paths refer to the directory where the tool is launched. So, if a custom path is specified, all commands must be executed in the directory where ASSURE has been executed, pointing to the proper target directory (see Chapter 5).

ASSURE generates the following elements in the output directory:

- a detailed log file `assure.log` that contains a detailed report of the tool execution. It contains all screen messages and other additional statistics.
- a subdirectory `hdl` that contains the obfuscated RTL code, i.e., the Verilog files generated after applying obfuscation to the input design;
- a subdirectory `scripts` that contains the scripts for formal verification with Synopsys Formality.

Obfuscation Option Selection. ASSURE can selectively apply the different obfuscation techniques:

- `--obfuscate-const <min-size>` applies constant obfuscation, where `min-size` represents the minimum bit-width of the constants to obfuscate. For example, with `--obfuscate-const 8`, all constants equal to or larger than 8 bits are obfuscated. With `--obfuscate-const 0`, all constants are obfuscated. Requires as many key bits as the total bit-widths of the constants.
- `--obfuscate-ops` applies operation obfuscation to arithmetic operations. Currently, the type of operation variants is fixed for each operator. For example, multiplications are always paired with dummy additions. Requires one key bit per operation.
- `--obfuscate-branch` applies control-flow obfuscation to ternary operators and `if-else` statements. Requires one key bit per control construct.

For example, it is possible to apply constant and control-flow obfuscation to the `i2cSlaveTop` module as follows:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop \
                                --locking-key ./input.key \
                                --obfuscate-const 0 --obfuscate-branch
```

ASSURE also features a single option (`--obfuscate`) to apply all obfuscation techniques. It is equivalent to the following string `--obfuscate-const 0 --obfuscate-ops --obfuscate-branch`.

3.2 Advanced use of Assure

Assure features a single option (`--obfuscate`) to apply all obfuscation techniques. However, the analysis is performed in deep-first search and obfuscation stops when the number of available key bits is reached. For example, when executing the following command with a 16-bit key:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop \
                                   --locking-key ./input.key --obfuscate
```

Assure uses only 16 bits even if the entire design requires 269 bits for obfuscation. So, Assure offers two alternatives to control the obfuscation: full obfuscation and selective obfuscation.

Full Obfuscation. It is possible to apply obfuscation beyond the key limit by specifying the option `--enable-key-reuse`. For example, Assure can obfuscate all elements of the previous design as follows:

```
$> python3 ${ASSURE_SRC}/assure.py -f ./file_list.f --top i2cSlaveTop \
                                   --locking-key ./input.key --obfuscate
    --enable-key-reuse
```

Even if only 16 bits are provided as input, the resulting design uses 269 bits that are obtained by replicating the input locking key as many times as needed.

Selective Obfuscation. It is also possible to exclude some portions of the original design from obfuscation to spare key bits that can be used for other parts of the design. This is achieved by specifying custom pragmas. For example, the previous design can be modified as follows:

File: i2cSlave

```
module i2cSlave (
  clk,
  rst,
  sda,
  scl,
  myReg0,
  myReg1,
  myReg2,
  myReg3,
  myReg4,
  myReg5,
  myReg6,
  myReg7
);
[...]
(* obfuscation_off *)
registerInterface u_registerInterface(.clk(clk), .addr(regAddr),
  .dataIn(dataToRegIF), .writeEn(writeEn), .dataOut(dataFromRegIF),
  .myReg0(myReg0), .myReg1(myReg1), .myReg2(myReg2), .myReg3(myReg3),
  .myReg4(myReg4), .myReg5(myReg5), .myReg6(myReg6), .myReg7(myReg7));
(* obfuscation_on *)
serialInterface u_serialInterface (
  .clk(clk), .rst(rstSyncToClk | startEdgeDet),
  .dataIn(dataFromRegIF), .dataOut(dataToRegIF), .writeEn(writeEn),
  .regAddr(regAddr),  .scl(sclDelayed[`SCL_DEL_LEN-1]),
  .sdaIn(sdaDeb), .sdaOut(sdaOut),
  .startStopDetState(startStopDetState), .clearStartStopDet(clearStartStopDet));
endmodule
```

In this case, the module `registerInterface` is excluded from obfuscation and the key bits can be used for the other module.

Due to a limitation of the Icarus Verilog parser, if the submodule or the subelement to be excluded is the last one of the module, the pragma `(* obfuscation_on *)` should be omitted.

Chapter 4

Interface of the Locked Design

This chapter describes the interface of the locked designs generated by ASSURE and how to unlock it. This process can be used to integrate the locked design into a larger chip. ASSURE adds an extra input port having the same dimension as the input locking key. This extra port is named `locking_key`.

Currently, ASSURE does not perform any verification on the names of the input/output ports. So, it assumes there is no port having the name `locking_key`.

For example, let us assume the design whose top module is named `i2cSlaveTop`. This design has the following interface:

File: `i2cSlaveTop.v`

```
module i2cSlaveTop (
  clk,
  rst,
  sda,
  scl,
  myReg0
);
[...]
endmodule
```

After executing ASSURE with a 16-bit locking key, the tool produces the corresponding output module that has the following interface:

File: `i2cSlaveTop.v`

```
module i2cSlaveTop_0_obf (
  clk,
  rst,
  sda,
  scl,
  myReg0,
  locking_key
);
[...]
  input [15:0] locking_key;
[...]
endmodule
```

where the suffix "`_0_obf`" has been added to the name of the top module and the interface has an extra input port named `locking_key`.

⚠ Currently, ASSURE does not implement any mechanism for key management, apart from key reuse. Hence, if any specific method is required, it must be implemented outside the locked module and connected to the module.

4.1 Design Organization

ASSURE uniquifies the input design, creating a different instance for each module. This allows the tool to apply different obfuscation schemas to each instance or to stop obfuscation when the maximum number of key bits has been reached. The log file reports statistics for each instance such as:

File: `assure.log`

```
[...]
INFO    : -----------------------------------------------------------------------
INFO    : | Original module                    = "serialInterface"
INFO    : | Obfuscated module                  = "serialInterface_3_obf"
INFO    : -----------------------------------------------------------------------
DEBUG   : | Total number of constants       = 79 CONSTANTS  / 210 BITS
DEBUG   : | Total number of operations      = 10 CONSTANTS  /  10 BITS
DEBUG   : | Total number of branches        =  6 CONSTANTS  /   6 BITS
DEBUG   : -----------------------------------------------------------------------
INFO    : | Number of obfuscated constants  =  4 CONSTANTS  /   7 BITS
INFO    : | Number of obfuscated operations =  0 CONSTANTS  /   0 BITS
INFO    : | Number of obfuscated branches   =  0 CONSTANTS  /   0 BITS
INFO    : -----------------------------------------------------------------------
INFO    : | Total number of bits used for obfuscation    = 7 BITS
DEBUG   : | Total number of bits needed for obfuscation  = 226 BITS
DEBUG   : | Current number of used key bits    = 16 BITS
INFO    : -----------------------------------------------------------------------
[...]
```

In this case, the tool reports the following information:

- the name of the original module `serialInterface` as well as the name of the newly-generated one `serialInterface_3_obf`.
- the total number of constants, operations, and branches that could be obfuscated in this specific instance as well as the corresponding number of key bits that would be needed.
- the number of constants, operations, and branches that have been effectively obfuscated in this specific instance as well as the corresponding number of key bits that have been used.
- the total number of bits used for obfuscating this module.
- the total number of bits that would have been needed to obfuscate this instance.
- the number of bits used for obfuscation so far (i.e., from the beginning of the analysis).

At the end of the analysis, the tool reports additional statistics, such as:

File: `assure.log`

```
[...]
INFO    : ---------------------------------------------------------------------------
INFO    : | registerInterface_2_obf    | File = "work/hdl/registerInterface_2_obf.v"
INFO    : | serialInterface_3_obf      | File = "work/hdl/serialInterface_3_obf.v"
INFO    : | i2cSlave_1_obf             | File = "work/hdl/i2cSlave_1_obf.v"
INFO    : | i2cSlaveTop_0_obf          | File = "work/hdl/i2cSlaveTop_0_obf.v"
INFO    : ---------------------------------------------------------------------------
INFO    : ---------------------------------------------------------------------------
INFO    : | Number of obfuscated constants  =   6 CONSTANTS /  16 BITS
DEBUG   : | Total number of constants       = 104 CONSTANTS / 244 BITS
DEBUG   : ---------------------------------------------------------------------------
INFO    : | Number of obfuscated operations =  0 OPERATIONS /  0 BITS
DEBUG   : | Total number of operations      = 14 OPERATIONS / 14 BITS
DEBUG   : ---------------------------------------------------------------------------
INFO    : | Number of obfuscated branches   =  0 BRANCHES /  0 BITS
DEBUG   : | Total number of branches        = 11 BRANCHES / 11 BITS
INFO    : ---------------------------------------------------------------------------
INFO    : | Total number of bits used for obfuscation   =  16 BITS
DEBUG   : | Total number of bits needed for obfuscation = 269 BITS
```

Specifically:

- the list of generated modules and the path to the corresponding Verilog files.
- the number of constants that have been obfuscated (and the corresponding number of bits that have been used), along with the total number of constants (and the corresponding number of bits that would be needed for complete obfuscation).
- the number of operations that have been obfuscated (and the corresponding number of bits that have been used), along with the total number of operations (and the corresponding number of bits that would be needed for complete obfuscation).
- the number of branches that have been obfuscated (and the corresponding number of bits that have been used), along with the total number of branches (and the corresponding number of bits that would be needed for complete obfuscation).
- the number of bits that have been used for obfuscation, along with the total number of bits that would be needed for complete obfuscation.

4.2 Design Unlocking

The module can be unlocked by providing the proper locking key. If the module is part of a larger design, the locking key signal must be propagated up to the module where the key is managed. For example, if the key is manage right outside the `i2cSlaveTop` module, it is possible to rewrite the original module as follows:

File: `design.v`

```
module i2cSlaveTop(
  clk,
  rst,
  sda,
  scl,
  myReg0);
[...]
  wire [15:0] locking_key;
  //module to generate the locking key;
  [...]
  //instance of the obfuscated module
  i2cSlaveTop_0_obf i0 (.clk(clk),
                       .rst(rst),
                       .sda(sda),
                       .scl(scl),
                       .myReg0(myReg0),
                       .locking_key(locking_key)
                       );
[...]
endmodule
```

Chapter 5

Formal Verification

Formal verification of the locked design generated by ASSURE can be run as follows in the directory where the tool has been executed:

```
$> fm_shell -f <out_dir>/scripts/<top_name>_0_obf_verify.tcl
```

The tool produces the scripts together with a TCL configuration file that specifies the values of the correct locking key. In this way, the formal verification process verifies that the original design is equivalent to the obfuscated design when the correct key is applied (i.e., when it is correctly unlocked).

⚠ Currently, ASSURE assumes that formal verification is performed with Synopsys Formality.

The directory `<out_dir>/scripts` also contains a link to the file where the locking key is specified. Hence, the user can manually change the values of the different key bits to evaluate the mismatches when a different key is applied.