

Reproducing the Experiments of “Leveraging the Benefits of Information Flow Tracking for Detecting Hardware Design Flaws”, FDL 2025

Srinidhi Rathnakar Ganiga
Institute of Embedded Systems
Hamburg University of Technology
Hamburg, Germany
srinidhi.rathnakar.ganiga@tuhh.de

Bernhard J. Berger
Institute of Embedded Systems
Hamburg University of Technology
Hamburg, Germany
bernhard.berger@tuhh.de

Goerschwin Fey
Institute of Embedded Systems
Hamburg University of Technology
Hamburg, Germany
goerschwin.fey@tuhh.de

1 Introduction

DuRTL is a tool for performing Dynamic Information Flow Tracking (IFT) for a given hardware design. An information flow from signal x to signal y exists if the value of signal y changes with a change in the value of signal x , keeping all other signals of the system constant. With a given hardware Verilog design and the corresponding Testbench, reflecting the feature testing of the design, DuRTL exports the modified design injecting tags and tag propagation logic. DuRTL performs a dynamic analysis of IFT by simulating this exported Verilog design and Testbench using the external tool Icarus Verilog [2]. Simulation results are stored in Value Change Dump (VCD) files and used for information flow analysis. The analysis of signal tags at any given time helps determine the information flow path between the signals or lists the dependent signals.

2 Background

The README contained in the DuRTL repository describes the installation and usage of DuRTL. In this documentation our aim is to provide an outline of how the tool can be used for the use-case described in the research work [1]. Reproducing the experiments described in the work [1] and providing an analysis of the experiment result with the help of a testcase **secureAccess_moduleTagging**, we provide a structured view of some of the features supported by DuRTL.

2.1 Terminologies

We need six inputs from the user, as follows:

- *ift* : Determines whether IFT is needed for the given design. This parameter is set to **true** to perform the IFT.
- *tag-size* : Represents number of possible tags during a simulation run. Tags are associated with signals at different timesteps. This parameter can be set with a value **32, 64, 128, etc**

- *tagged_module_name* : Refers to the module whose signals information flow is needed. Set the parameter with the **name of the module whose signals need to be analyzed in the design**.
- *Verilog_file* : Refers to the Verilog design file. Set the parameter with the **relative path to Verilog file**.
- *testbench_file* : Refers to the testbench of the design file. Set the parameter with the **relative path to the testbench file**.
- *json_file* : Refers to the json generated for the given Verilog design file. The README describes the necessary steps to generate a JSON file for a given Verilog file. Set the parameter with the **relative path to the generated json file**.

2.2 Working

For a given Verilog design, we parse the json file to obtain all the modules, cells, and ports associated with the design. This parsed information is used to generate a flattened graph of the design with tags associated in the exported design. The DuRTL documentation covers the necessary details on the supported interfaces and signal tagging to aid IFT. The interfaces that are not introduced in the README – and are directly used in the testcase – *get_timesteps_per_simulation_run_internal_module_tagging*. The remaining interfaces that support internal module tagging can be seen in *testbench.cpp* file. A more generic DuRTL IFT use-case involves tagging the inputs of the Top Module of the design and analyzing the information flow of these input signals through the design to the intermediate signals and output signals of the design. Setting *tagged_module_name* parameter and using the interface *get_timesteps_per_simulation_run_internal_module_tagging* alternative to the interface *get_timesteps_per_simulation_run*, we can export the design with tagged outputs of *tagged_module_name* module instead of inputs of the Top Module of the given design.

3 Replicating the Results

With understanding of the necessary inputs from the earlier section, we present how the results from [1] can be replicated in this section. For the given prototype design, tracking the information flow path of *SecretKey* is relevant.

- *SecretKey* is generated from the module *Assigner Core* [1]. Therefore, it is chosen as the internal module whose signals information flow we are interested in i.e. *Assigner Core* is set as *tagged_module_name*.
- Tagging the output signals of *Assigner Core* module, we can see the information flow path for *SecretKey*. *SecretKey* refers to *assignerCoreSec_grantAccessUsecase_o* as seen in the testcase **secureAccess_moduleTagging**.
- Signal tag analysis post design export from DuRTL is implemented after extracting the tag values for the signal using the interface *instance.get_tag_values*.
- We see from the testcase **secureAccess_moduleTagging**, *SecretKey* propagated to *RSA Core* and *AES Core* represented as *RSACoreSec_secretData_i* and *AESCoreSec_secretData_i* respectively have the same tags as *SecretKey* generated from *Assigner Core* represented as *assignerCoreSec_grantAccessUsecase_o*. This helps us to conclude that there exists an information flow path or *SecretKey* leakage from the generated *Assigner Core* to the lower privilege *AES Core*.

- Performing the exact signal tag analysis for the **Fixed Design** as seen in test-case **secureAccess_moduleTagging_Fix**, we can see that no tags from *assigner-CoreSec_grantAccessUsecase_o* are seen in *AESCoreSec_secretData_i*, i.e., no information flow path or *SecretKey* leakage to *AES Core*.

References

- [1] S. Ganiga, B. Berger and G. Fey, "Leveraging the Benefits of Information Flow Tracking for Detecting Hardware Design Flaws" in 2025 Forum on specification and Design Languages, IEEE Xplore Proceedings
- [2] "Icarus verilog." [Online]. Available: <https://github.com/steve icarus/iverilog>