Checking security path with formal verification tool: new application development

Julia Dushina (Julia.Dushina@st.com) STMicroelectronics, UK, Bristol
Saumil Shah (Saumil@cadence.com) Cadence, UK, Bracknell
Joerg Mueller (jmueller@cadence.com) Cadence, Germany, Munich
Vincent Reynolds (vincentr@cadence.com) Cadence, UK, Bracknell

Abstract: *This paper describes how a new formal tools application was developed resulting from an actual need of STMicroelectronics chip development. The new formal application is checking that sensitive data can't leak towards unwanted interfaces of design.*

Introduction

In many industries such as credit card chips, set-top-box industries, etc. the data security is one of the main concerns of manufactures as the failure to protect sensitive information results in huge financial loss and sometime loss of the business altogether due to damaged reputation [1, 2].

In STMicroelectronics we were concerned about the security of sensitive data in a chip developed by the Set-top-box division. The sensitive information in our case is the keys used to decrypt data stream provided to consumers' cable and satellite end point. Checking that sensitive data is not accessible from external interface requires checking that the path from sensitive data (key store) to external interface is secure, i.e. sensitive data can't flow to the external interface. A path from sensitive data store to external interface is further called a security path.

Following exploration of available on the market methodologies and techniques, our formal verification tool supplier (Cadence) was asked to propose a solution, which was accepted by ST using quality to price criterion. The proposed solution is described below.

Design description

Due to security reason all the details of the design can't be revealed, so general functionality of a simplified example is shown in Figure 1 below.

The design is manipulating keys which are used to encrypt/decrypt satellite or cable data stream. Whereas all three design interface (denoted in Figure 1 as Interface A, B and C) can write keys, only one interface A can read keys back. Moreover, this interface A can only read a key if allowed so by an adjacent rules block. Rules block will take a key's address as input and will return "allowed/not allowed for reading" signal back to the DUT. In addition to the key store and rules block, the design is also writing data to system memory and system memory can provide data back to the design.

So, the task was to prove that no accidental leakage of the keys occurs towards the interfaces B and C and the keys are only permitted to appear on the interface A when approved by the rules block.

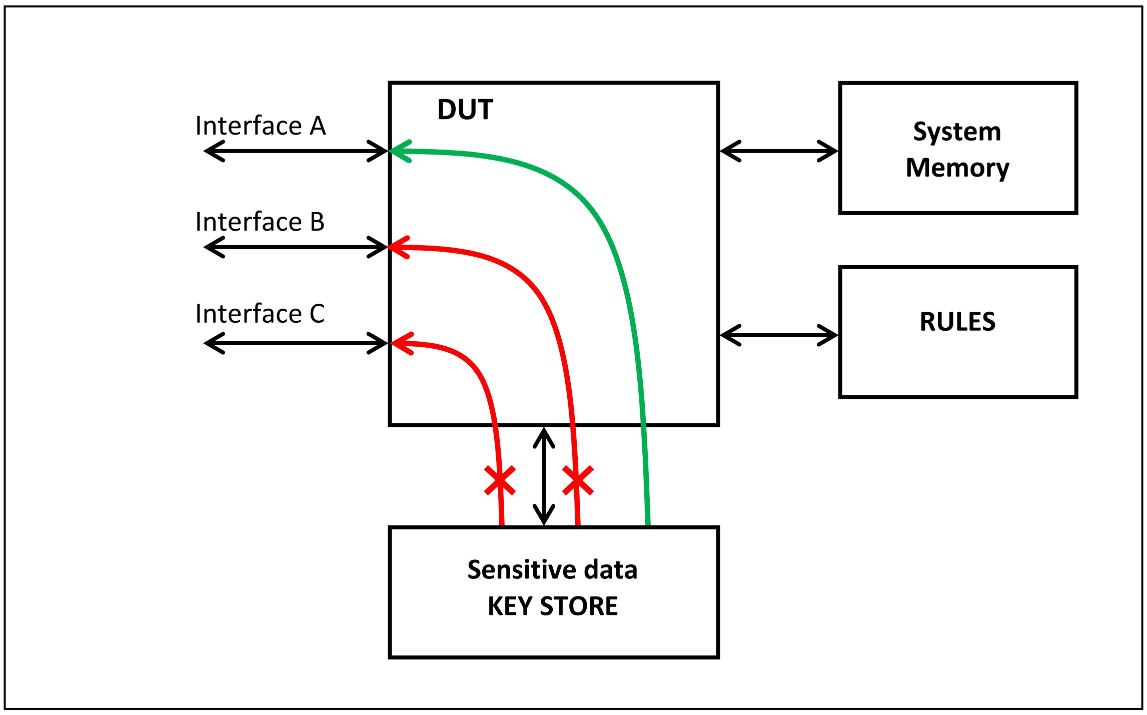


Figure 1: Block diagram of secure device

Explored solutions

When searching for an answer to the problem, three possible solutions were explored.

Symbolic approach

This approach uses an abstract constant data value called symbol that would travel from key store towards external borders. A property is using the symbol for checking appearance on interface C. In order for the symbol to be sourced only from the sensitive key storage, it was prohibited with the help of constraints to enter on any other interface, especially on the system memory interface as it directly provides data to the interfaces A, B and C.

The set of constraints and main property would look in this approach as outlined below. The PSL flavour of interactive properties (a new Tcl runtime feature provided by the tool [6]) is used as formal language in the following example to write security path property for the interface C. An assumption in the example below is that every interface has forward (*data*) and returned data (*return_data*) and that all data buses of the design have the same width:

- Define a new signal *symbol* and constrain it to be rigid during proof:
  constr -add -rigid symbol
- Prohibit all other interface to have the data equal to symbol:
  constr -add -inter { interface_a _data != symbol} -name const_data_on_a
  constr -add -inter { interface_b _data != symbol} -name const_data_on_b
  constr -add -inter { interface_sys_mem _data != symbol} -name const_data_on_sys_mem
- Add a cover showing the symbol can enter the design at the secure interface
  assert -add -inter { keystore_r_data == symbol } –cover -name cover_symbol_on_key_store
- Write the main property that proves the absence of the data on the interface C:

assert -add -inter { (interface_c_return_data == symbol) } -cover -name cover_no_symbol_on_c
assert -add -inter { ! (interface_c_return_data == symbol) } -name check_no_symbol_on_c

Please notice that we can prove the absence of unwanted data on the interface C by writing either a cover or an assertion. In first case we expect the cover to fail proving that there is no path (no coverage can be found) between the key store and the interface C and therefore the interface C return data can never be equal to the symbol. In the second case we are negating the expression used in the cover statement. And in this case we expect the assertion to pass.

For the interface A more constraints are needed as keys can be read by the interface A if approved by the rules block. So the task becomes to prove that keys can't be read if the rules prohibit it. Thus, the additional constraint stipulates that if a key is read by the interface A, the rules block returns "not allowed for reading" signal for that key defined by the key address:

- Define a new signal *address_symbol* and constraint it to be rigid during proof:
  constr -add rigid address_symbol
- Prohibit reading a key of the above address whereas all other keys may be read:
  constr -add -inter { (interface_a_req == 1 and interface_a_address == address_symbol) |=> sys_memory_allowed_for_reading_signal == 0} -name constr_no_reading_rules

A new symbol is introduced to represent key address when reading a key on the interface A. We took advantage of the fact that there is no re-timing between the interface A request and request on the rules block. The "allowed_for_reading" signal returns its value the next clock cycle. Thus, the main properties for the interface A becomes as follows:

assert -add -inter { (interface_a_return_data == symbol and interface_a_address == address_symbol)) } -cover -name cover_no_symbol_on_a
assert -add -inter { ! (interface_a_return_data == symbol and interface_a_address == address_symbol)) } -name check_no_symbol_on_a

The provision of constraints to prevent the observed symbol from entering the system at other, legal interfaces may seem as additional effort. However, it is this iterative process of analyzing and eliminating one legal data flow in the IP after the other, that increased the confidence and trust in the security solution when eventually no path is reported.

Miter approach

Two instances of the same design are placed side by side and all corresponding inputs except for the key store output are tied together forming a Miter [3]. The key store outputs are described in a cover property to be different from each other, so we know that it can be the origination of a discrepancy. An assertion then checks that corresponding outputs of external interfaces are always equal. As the only variable between the two instances comes from the key store interface then the only way a difference can be observed between two corresponding outputs is if a path exists between the key store and the output being verified. If the assertion passes then the path is secure, if it fails then a security violation is exposed.

The code of the approach applied to the interface C is as follows:

- In a testbench tie all inputs but from the key store together as shown in Figure 2
- Cover the inputs from the key store can be different for two instances:
  assert -add -inter { (key_store_input_inst1 != key_store_input_inst2)) } -cover -name cover_different_key_store_inputs
- Write the main property that proves returned data on the interface C are always the same:
  assert -add -inter { (interface_c_return_data_inst1 == interface_c_return_data_inst2)) } - name same_data_on_c

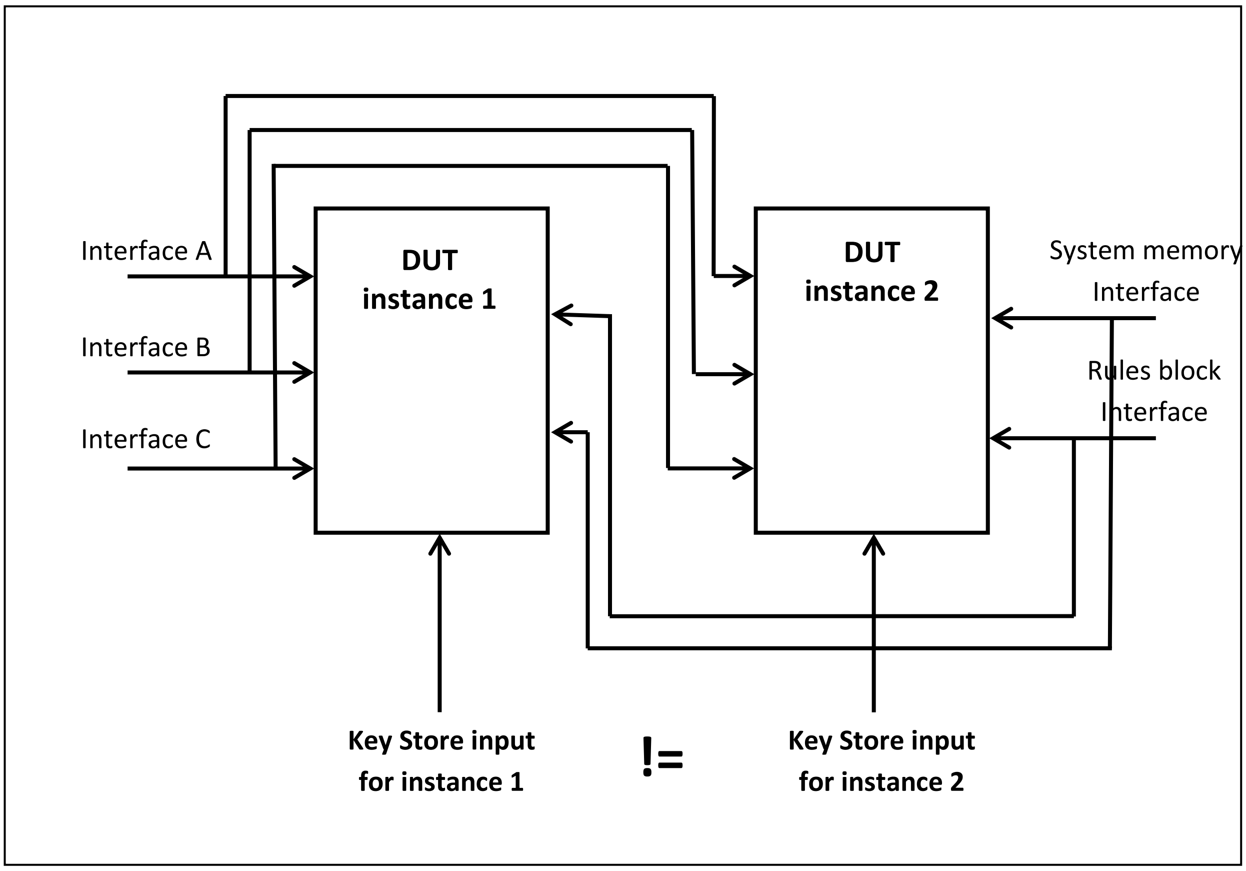


Figure 2: Miter construction

Please notice that in place of tying inputs of the two instances in testbench, they can be constraint instead:

- Constraint all input of the DUT but inputs coming from the key store be the same for two instances:
  constr -add -inter { input1_inst1 == input1_inst2} –name const_input1
  constr -add -inter { input2_inst1 == input2_inst2} –name const_input2
  . . .
  constr -add -inter { inputN_inst1 == inputN_inst2} –name const_inputN

We did not find a noticeable difference in the 2 ways to connect the inputs.

In contrast to the previous approach, using the Miter approach the origin of the observed value is unique by construction – it can only come from the key store. Thus there is no need to create constraints for preventing values originating from other inputs. While this approach is simpler to set up, it lacks the insights and learning during environment creation, resulting in reduced confidence in the environment.

X-Propagation approach

This approach uses already available Cadence capacities to propagate “X”s in designs. In this approach the output of the key store is explicitly assigned to “X” and a property is written that proves that “X” would never appear on the external interface [4, 5].

Conceptually this is exactly the same approach as the Miter approach described before. Just instead of tracing a unique value modelled as discrepancy we trace a unique value modelled as “X” in the tool.

The code of this approach applied to the interface C is as follows:

- Inject “X” at the key store return data:
  constr -add -inter { (!keystore_csn && keystore_wen) |=> (keystore_r_data[7:0] === 8'hXX) || (keystore_r_data[7:0] !== 8'hXX) } -name inject_x_at_keystore_r_data
- Write the main property that proves returned data on the interface C are never “X”:
  assert -add -inter { (interface_c_return_data !== 8’hXX)) } -name check_no_x_on_c

Current solution to the problem

At the end of the exploration, the symbolic approach was used to prove that there is no path from the key store towards three external interfaces. Three sets of environment and main assertions were created, one for each interface.

Blackboxing

As the design is very complex irrelevant parts of the design are black boxed in order to conclude on the assertions. Blackbox outputs become new primary inputs to the design. It must be decided whether a black box output can be the new origin of a secret data (in case it would transport data arriving at its inputs) or not in the same way as for primary design inputs. It was concluded not to consider them as origins, because black boxing is applied selectively and dependency between black boxed blocks and each interface is considered. For example for the interface A, only the modules which affect the data flow from the interfaces B and C towards the key store are black boxed but not the modules which affect the data flow from the interface A. Likewise appropriate blackboxing is applied when checking interface B and C. Figure 3 shows the idea.

Results

The security path verification was successfully applied on three different paths, resulting in three properties to be verified in total, guaranteeing security of secret keys on the chip.

At least one security hole was found while checking the security paths which are now all fixed. The main problem was keeping keys written to the key store on the data bus which was then propagated back to the requesting interface during protocol acknowledgment phase.

At the end we were able to conclude all properties and guarantee security on all paths with IEV. The total runtime for the assertion was varying from 5 to 30 minutes using a 3 GHz Quad-core Xeon machine with 24 GB of RAM.

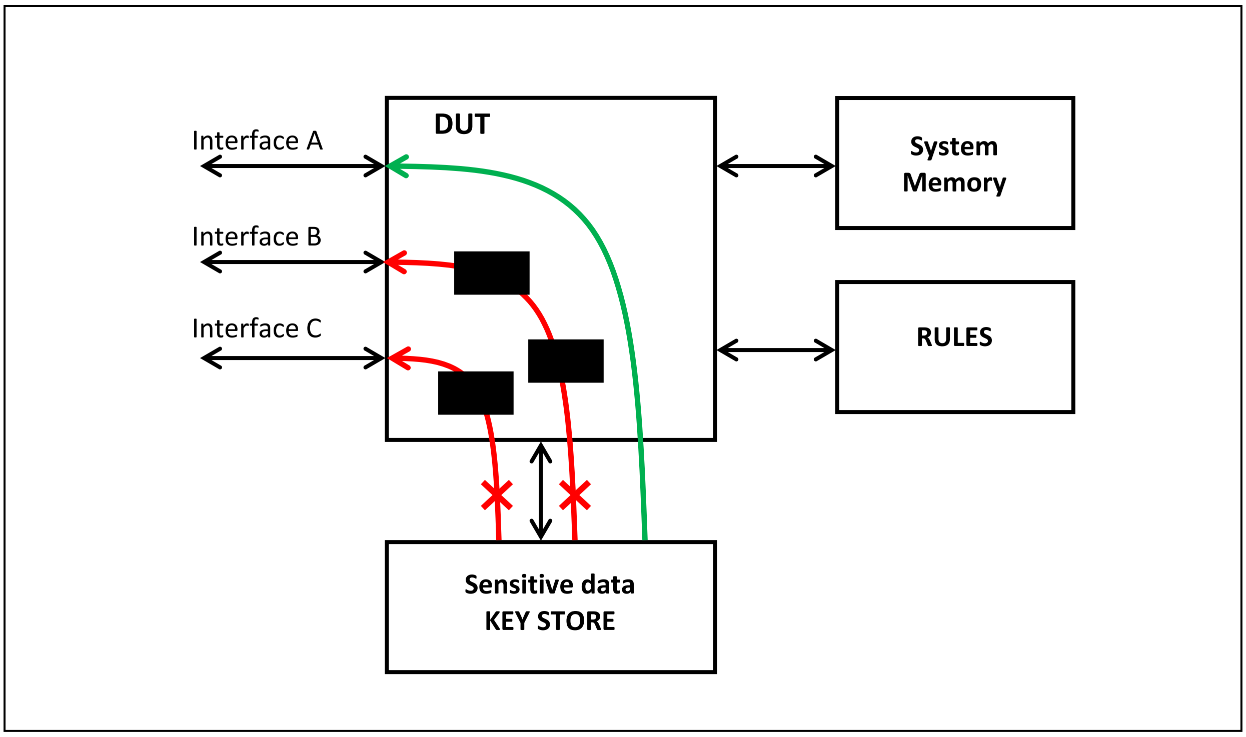


Figure 3: Blackboxing applied on the device

New formal tools application

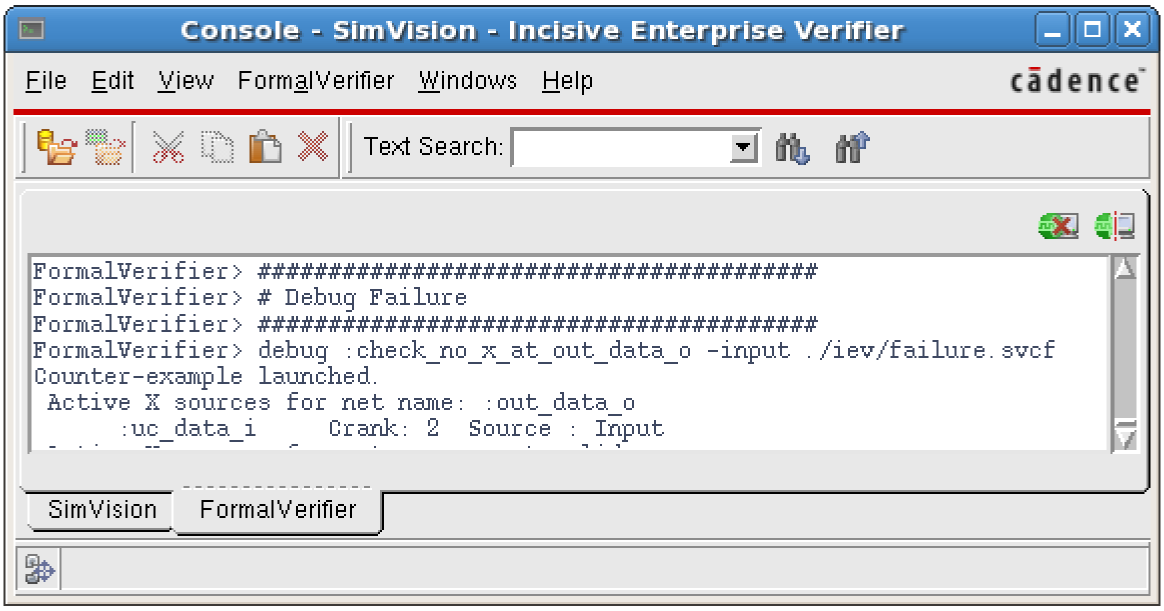
Following this experiment Cadence created a prototype of a new formal tool application called “Security Leakage Verification Solution” incorporating the learning and experiences from our collaboration. It is automating the setup by providing new Tcl commands, improving the productivity by new debugging capabilities dedicated to this application and improving performance of the engines specifically for these setups.

An example command sequence to setup the verification for a single path from the keystore to the interface C is shown below.

```
check_no_path keystore_r_data  interface_c_return_data
prove
```

It will take care about setting up the tool, creating the necessary constraints, covers and checks and running the formal analysis. So the user is not required to write any property code directly, rather he provides a high level description of the desired or undesired paths in his design.

In case of a security leakage the tool provides a formatted waveform (Figure 5) and a message showing the root origin (Active X sources in Figure 4). That is also extremely useful while creating the environment for understanding the false paths that needed to be eliminated. The examples below are taken from a demo design rather than the ST design.


Console - SimVision - Incisive Enterprise Verifier
File Edit View FormalVerifier Windows Help
cādence
Text Search:
SimVision
FormalVerifier

```
FormalVerifier> ##########################################
FormalVerifier> # Debug Failure
FormalVerifier> ##########################################
FormalVerifier> debug :check_no_x_at_out_data_o -input ./iev/failure.svcf
Counter-example launched.
 Active X sources for net name: :out_data_o
      :uc_data_i     Crank: 2  Source : Input
```

Figure 4: Debugging a security leakage in IEV showing the root cause

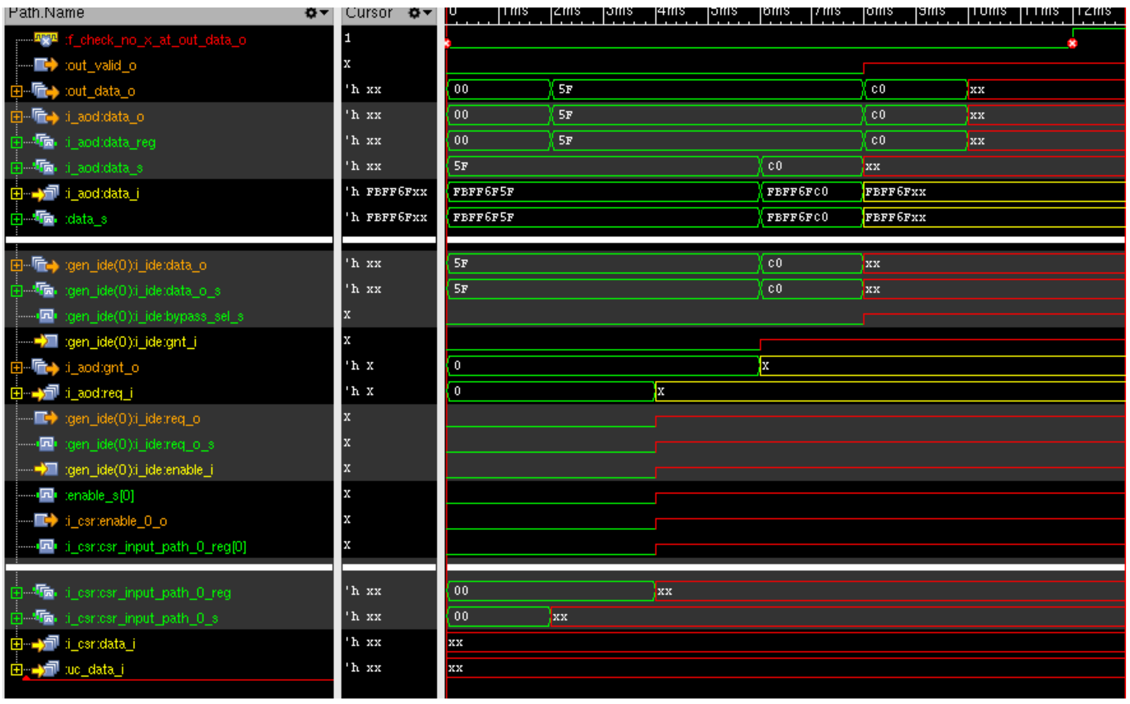


Figure 5: Waveform showing a security leakage

Conclusion

From ST perspective it was provided with robust and affordable solution to the security path problem which is otherwise very difficult to tackle. This solution fits nicely with other tools range from the same supplier making its use familiar and therefore easier as all tools have similar interface and features.

For Cadence this experience allowed to create a new formal tool general purpose security path application thus easing the task of proving security path feature and providing crucial confidence in the security hardware.

The resulting “Security Leakage Verification Solution” is automating the creation of the environments by providing built-in capabilities for creating the models and properties required for guaranteeing absence of data path between secure and non-secure interfaces.

References

[1] Paul Kocher, Joshua Jaffe, and Benjamin Jun, Differential power analysis”, Springer-Verlag, 1999, pp 388-397

[2] Michael Tunstall, “Attacks on Smart Cards”, http://www.cs.bris.ac.uk/home/tunstall/presentation/AttacksonSmartCards.pdf

[3] A. Jain, V. Bopanna, R. Mukerjee, J. Jain, M. Fujita, M. Hsiao, “Testing, Verification and Diagnosis in the Presence of Unknowns”, 18th VLSI Test Symposium 2000, pp 263-269.

[4] D. Brand, R. A. Bergamaschi and L. Stok, “Be Careful with Don’t Cares,” DAC’95, pp.83-86.

[5] R. A. Bergamaschi, D. Brand, L. Stok, M. Berkelaar, and S. Prakash, “Efficient Use of Large Don’t Cares in High-level and Logic Synthesis,” ICCAD’95, pp. 272-278.

[6] Incisive Enterprise verifier XL User Guide. http://support.cadence.com, 2013. Product Version 13.1.