Hardware Security Module
SIM - CIATEQ

Documento de Pruebas de Proyecto

Módulo hardware de criptografía ligera orientado al internet de las cosas

CIATEQ

Versión 1.0a

Histórico de Revisiones

Date	Descripción	Autor(es)
xx/xx/xxxx	<Descripción>	<Autor(es)>
xx/xx/xxxx	▪ item; ▪ item;	<Autor(es)>

ÍNDICE

1. Introducción

1.1. Vista General del Documento

En este documento se redacta la información necesaria para realizar las pruebas al módulo hardware para seguridad basado en el estándar [1].

- **Requisitos funcionales** - Lista de todos los requisitos funcionales.
- **Requisitos no funcionales** - Lista de todos los requisitos no Funcionales.
- **Dependencias** - Conjunto de dependencias de IP-cores previstos.
- **Notas** - Lista de notas presentadas en el documento.
- **Referencias** - Lista de todos los textos referenciados en el documento.

1.2. Definiciones

Término	Descripción
Requisitos Funcionales	Requisitos que hacen funcional al sistema, son las capacidades que debe tener el sistema entregado.
Requisitos Técnicos	Requisitos del sistema que definen características referentes a técnicas, algoritmos, tecnologías y especificidades de los requerimientos funcionales.
Requisitos No Funcionales	Requisitos de los módulos entregables. Se refieren a las capacidades no funcionales del sistema como un todo y que especifican necesidades del usuario final.
Dependencias	Requisitos de reuso de IP-cores, describiendo las funciones que cada uno de estos módulos debe realizar.

1.3. Acrónimos y abreviaciones

Sigla	Descripción
FR	Requisito Funcional
TR	Requisito Técnico
NFR	Requisito No Funcional
D	Dependencia

1.4. Prioridades de los Requisitos

Prioridad	Característica
Importante	Requisito para que el sistema sea entregado.
Esencial	Requisito que debe ser implementado para que el sistema funcione.
Deseable	Requisito que no compromete el funcionamiento del sistema.

2. Requisitos Funcionales

En un sistema HSM, un controlador maestro envía peticiones de servicios continuamente al HSM, entonces, el HSM responde a dichas peticiones con servicios de seguridad. Debido a que hay muchas solicitudes del controlador maestro, el HSM debe responder a las solicitudes muy rápidamente. Para este propósito, el microcontrolador y otros módulos FPGA deben estar altamente optimizados [2].

En esta etapa del desarrollo del HSM, no se determina cómo tomará forma el progreso del software que utilizará los servicios del HSM, los requisitos existentes definen las funcionalidades del sistema y los algoritmos que se implementarán en FPGA para el cifrado, el *hashing*, firma digital y la generación de llaves.

2.1. Requisitos Funcionales

[FR1] Cada llave debe ser usada por una sola función criptográfica

DescripciÃşn: The HSM ensures that each cryptographic key is only used for a single cryptographic function and only for its intended purpose.

Nivel de Prioridad: **Importante**

[FR2] Cálculo de resumen (*hash*) de mensajes

DescripciÃşn: permite la generación y verificación firmas digitales hash y adicionalmente HMAC.

Nivel de Prioridad: **Importante**

[FR3] Cifrado asimétrico

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

[FR4] Cifrado simétrico

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

[FR5] Cálculo de números pseudo-aleatorios

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

[FR6] Proveer un contador monotónico

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

[FR7] Creación de llaves internamente

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

2.2. Requisitos Técnicos de los Requisitos Funcionales

[TR1] Requisitos Técnicos de FR1

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

[TR2] Requisitos Técnicos de FR2

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

[TR3] Requisitos Técnicos de FR3

- El algoritmo utilizado es AES-128

[TR4] Requisitos Técnicos de FR4

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

[TR5] Requisitos Técnicos de FR5

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

[TR6] Requisitos Técnicos de FR6

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

[TR7] Requisitos Técnicos de FR7

- Los datos que serán utilizados llegan bloque a bloque al HSM
- El algoritmo utilizado es SHA-256
- El *hash* se envía a una dirección especificada previamente

3. Requisitos no Funcionales

[NFR1] Nombre del Requisito

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

[NFR2] Nombre del Requisito

DescripciÃşn: Descripción breve y objetiva.

Nivel de Prioridad: **Importante**

4. Dependencias

[D1] Nombre del IP-*core*

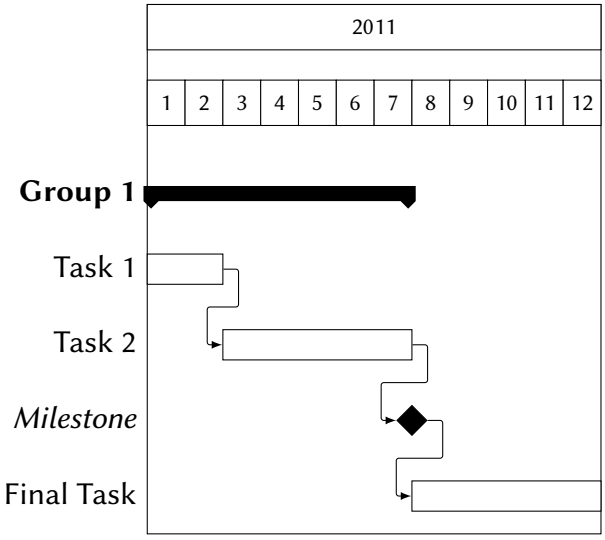
Descripción breve y objetiva del IP-*core* y referencia al documento.

[D2] Nombre del IP-*core*

Descripción breve y objetiva del IP-*core* y referencia al documento.

5. Componentes del sistema

6. Planeación



7. Vista general de UVM

The Universal Verification Methodology (UVM) that can improve interoperability, reduce the cost of using intellectual property (IP) for new projects or electronic design automation (EDA) tools, and make it easier to reuse verification components is provided. Overall, using this standard will lower verification costs and improve design quality throughout the industry. The primary audiences for this standard are the implementors of the UVM base class library, the implementors of tools supporting the UVM base class library, and the users of the UVM base class library.

Role of each testbench element is explained below, UVM test The test is the topmost class. the test is responsible for,

configuring the testbench. Initiate the testbench components construction process by building the next level down in the hierarchy ex: env. Initiate the stimulus by starting the sequence. UVM Environment Env or environment: The environment is a container component for grouping higher level components like agent's and scoreboard. UVM Agent UVM agent groups the uvm components specific to an interface or protocol. example: groups the components associated with BFM(Bus Functional Model). The components of an agent are,

UVM SEQUENCE ITEM The sequence-item defines the pin level activity generated by agent (to drive to DUT through the driver) or the activity has to be observed by agent (Placeholder for the activity monitored by the monitor on DUT signals). UVM DRIVER Responsible for driving the packet level data inside sequence item into pin level (to DUT).

UVM SEQUENCE Defines the sequence in which the data items need to be generated and sent/received to/from the driver. UVM SEQUENCER Responsible for routing the data packet's(sequence item) generated in sequence to the driver or vice verse. UVM MONITOR Observes pin level activity on interface signals and converts into packet level which is sent to components such as scoreboards. UVM Scoreboard Receives data item's from monitor's and compares with expected values.

expected values can be either golden reference values or generated from the reference model.

8. Herramienta de pruebas *toolchain*

9. Pruebas unitarias

10. Pruebas de integración

11. Pruebas de sistema

12. Análisis con herramientas *Lint*

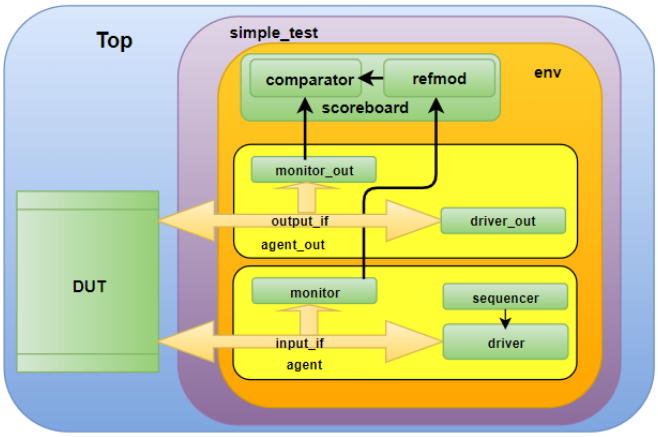
12.1. Verilator

What Verilator Does Verilator is invoked with parameters similar to GCC or Synopsys's VCS. It "Verilates"the specified synthesizable Verilog or SystemVerilog code by reading it, performing lint checks, and optionally inserting assertion checks and coverage-analysis points. It outputs single- or multi-threaded .cpp and .h files, the "Verilatedçode.

The user writes a little C++/SystemC wrapper file, which instantiates the "Verilated"model of the user's top level module. These C++/SystemC files are then compiled by a C++ compiler (gcc/clang/MSVC++). The resulting executable performs the design simulation. Verilator also supports linking its generated libraries, optionally encrypted, into other simulators.

Verilator may not be the best choice if you are expecting a full featured replacement for NC-Verilog, VCS or another commercial Verilog simulator, or if you are looking for a behavioral Verilog simulator e.g. for a quick class project (we recommend Icarus Verilog for this.) However, if you are looking for a path to migrate synthesizable Verilog to C++ or SystemC, and your team is comfortable writing just a touch of C++ code, Verilator is the tool for you.

go to https://www.veripool.org/projects/verilator/wiki/Installing



13. Referencias

[1] IEEE, "Ieee guide for software verification and validation plans," *IEEE Std 1059-1993*, pp. 1–87, April 1994.

[2] M. Wolf and T. Gendrullis, "Design, implementation, and evaluation of a vehicular hardware security module," in *Information Security and Cryptology - ICISC 2011* (H. Kim, ed.), (Berlin, Heidelberg), pp. 302–318, Springer Berlin Heidelberg, 2012.