Is Your System's Security preserved?

Verification of Security IP integration

Predrag Nikolic
Veriest Solutions
21000 Novi Sad, Serbia
predragn@veriests.com
Tel: +381643667559

***Abstract*-Most modern SoCs contain a HW Security block. The main task for this block is to protect the entire system from outside attacks. Without a Security block, the system would be open to malicious attacks that could lead to undesired behavior and results. Since such Security blocks became very complex, many companies do not develop them directly, but rather opt to license third-party IPs from companies specialized in this field. Once the IP is integrated to the RTL, it is essential to verify the correctness of the integration, i.e., to confirm that all Security features are preserved and available. This paper will focus on describing a Functional verification approach for confirming that third-party Security IP was properly integrated to the system, by using a SV/UVM coverage driven methodology. The example used in this paper is related to SoCs for the Automotive Industry - Autonomous vehicles.**

I. INTRODUCTION

All systems have vulnerabilities and things that must be protected. In the context of Autonomous vehicles, the Security objectives are Sensor authentication, Communication authentication, SW image integrity and authenticity, Secure key storage and confidentiality, and Sensor data integrity. In short, the device's Security system needs to prevent a pirates` attempts to replace sensor data, fake data injections, SW's modifications, and to detect jammer and spoofing attacks. Also, one of the main goals is to protect against accessing the device's secret information, e.g., keys and certificates.

A Security IP is an embedded secure element that provides the Root-of-Trust for the complete SoC. As a block, the Security IP must boot with a minimum set of instructions contained in the Boot ROM, must check data authenticity and physical integrity of its own components, and must provide services requested by the SoC's CPU through the Mailbox interface after the boot sequence completes. Such services include crypto services (data encryption and decryption), key management, and true-random number generation. The Security Root-of-Trust and Security Crypto engines reside inside the Security IP, but they have an integration impact on each of the SoC's subsystems. Some of the security mechanisms are implemented in SW, and others are HW oriented.

Security IP is delivered by a third-party vendor with assurances of its correct behavior at the block level, and that it is ISO 21434 compliant [1]. Such assurances must be backed up by verification results and coverage database documentation. However, to make sure that all previously mentioned security objectives are fulfilled, block level verification alone is not enough, because many vulnerabilities are introduced at the system level. Therefore, system level verification is also needed. This paper will describe in detail all the system level verification steps that need to be taken before sign-off.

Statistics data shows [2] that the number of HW vulnerabilities grows exponentially year after year. This is explained by the increase of security support in HW, and the complex interaction between Security SW and HW, which leads to very complex system level verification. Not only are the implementation and verification prone to errors, but so is the architecture of the Security HW itself. The same source mentions specific Security logical/architectural bugs that were recently exposed that are remotely exploitable (i.e., do not require local access), which often require HW changes that can potentially cause enormous negative business impact.

Due to its complexity, Security IP verification - at both block and system levels - usually requires high effort and cost to complete. The paper in [3] explains the process and methodology that IP providers and integrators can follow to produce more secure products, but still has limitations: it is difficult to support growth, it was created from a relatively small sample of companies, and it does not address accuracy, completeness, and the quality of the

collateral, as described in [4]. The Security Annotation for Electronic Design Integration (SA-EDI) Standard evolved from the previously mentioned paper, and it provides complete guidance for IP providers to identify security concerns and either mitigate them in their IP or disclose to the integrator to address them at their level [5]. A description of another approach [6] has also been proposed that uses partly Functional verification, but also relies on Formal verification to make sure that the Security verification is secure.

The Verification approach used in this paper does not directly rely on any of the above-mentioned solutions, but rather it implements a different approach that is based on detecting the gaps between Block level and System level verification, and finding the simplest, yet sufficient solution. The approach is based on the methodology used in most of semiconductor companies, with the language in highest demand – constrained-random coverage driven Functional verification using SV/UVM [7]. The Verification environment is built using a UVM methodology approach, and therefore, we believe, there is no need to describe the environment in detail here. Therefore, only the Security specific parts of the environment will be described.

II. Verification of Security IP integration

The most important step in the planning process is to fully identify and understand the gaps between Block level and System (integration) level Security verification. One of the most important and most sensitive topics is the difference in content of the Boot ROM. In addition, the connectivity, timing and accessibility between Security IP on one side, and PMU, main CPU, Memories and other SoC blocks on the other side, cannot be verified at the Block level. Debug capabilities are also often overlooked and underestimated in Block level verification. Lastly, the performance of the integrated Security IP also heavily depends on the rest of the SoC, as well as on certain defined use cases. On the other hand, block level verification contains many scenarios that are sufficient to be verified only at block level, and to reduce the effort and cost, should not be re-verified at system level. For instance, side channel attacks are an interesting topic - but these scenarios are already sufficiently covered at the Block level, so there is no added value in double checking them in this specific SoC level. Thus, they are not in the scope of this paper.

Figure 1 presents a simplified Security system overview, displaying key elements needed for understanding the Verification flow. The Security IP is composed of the following blocks:

1) Mailbox Slave block is used for providing instructions and data to Security CPU,
2) DMAs are Masters used to provide data to the SoC,
3) Security CPU, with its associated IRAM, DRAM and Boot ROM memories,
4) Debug unit – to be able to access complete Security IP's memory space under specific circumstances,
5) Crypto engines, True Random Number Generator (TRNG), One-Time Programable (OTP) memory, and Digital sensors – blocks needed for providing security specific services.

Security IP is connected to the rest of the SoC via the System interconnect unit. The most important connections are to the main CPU and Memories (in this example, Non-volatile memory (NVM)).

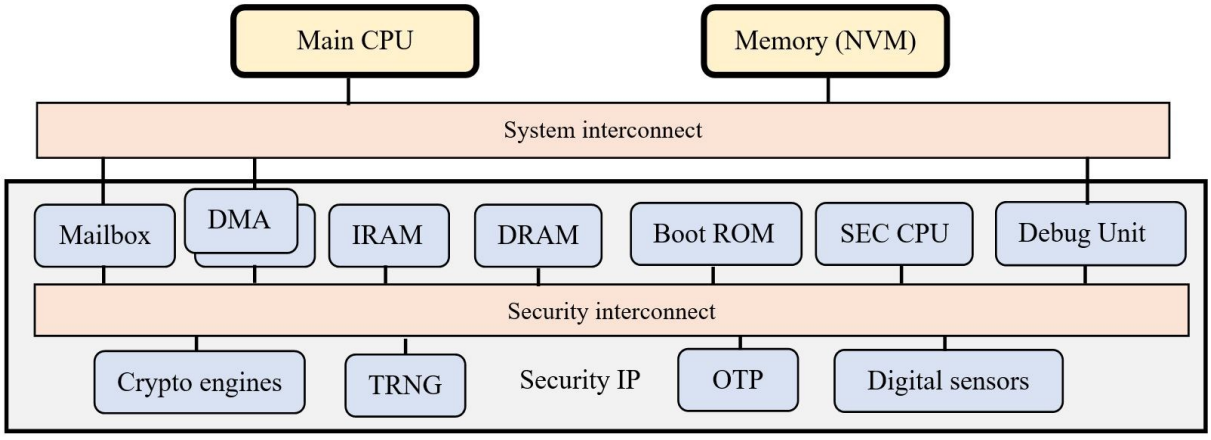

Figure 1. Security subsystem – block diagram

The Verification approach can be distributed into several groups. Each group can be represented with one or more tests. Table 1 lists the minimal set of tests needed for verification of the Security IP integration. Each of the tests will be described in more detail in the following chapters.

TABLE I
MINIMAL PROPOSED TEST PLAN

Test name	Description
non_secure_boot_test	Complete scenario for Non-secure boot.
secure_boot_test	Complete scenario for Secure boot. This scenario includes stimuli for each of the Debug units that get locked in Secure mode.
trng_test	TRNG service request, used for simulating the communication between Security IP and the rest of the system
crypto_service_test	Crypto service request, used for simulating the communication between Security IP and the rest of the system
debug_inf_non_secure_test	Scenarios for accessing Security IP's Debug blocks in Non-secure mode of operation
debug_inf_secure_test	Scenarios for accessing Security IP's Debug blocks in Secure mode of operation
hw_alarm_interrupts_test	Scenarios for setting HW alarm and interrupts
performance_path_1_test	Scenario for performance test, for verification of one critical path. Each critical path should have separate test.

A. Boot flows – Secure and Non-secure modes of operation

The Security IP Boot flow starts once the block is out of reset. When used at the SoC level, this boot flow is responsible for performing internal configuration, as well as part of the power-up sequence of the SoC. In Block level verification, this flow performs only internal configuration (no power-up sequence needed for block level). This power-up difference in the Boot ROM flow is one of the main differences in Security IP verification at the integration SoC level vs. the Block level.

In general, this means that the Boot ROM code cannot be simply used as provided by IP provider, at least not the same code that was used in Block level verification. At Block level verification, there is no interest in creating Boot ROM code that performs external transactions. In addition, the IP provider is not familiar with the SoC power-up flow and address space. Therefore, the Boot ROM code needs to be specifically generated for the purpose of SoC power-up.

One solution is to provide all necessary information to the IP vendor, so they can generate the specific Boot ROM code that can be used out-of-the-box for this SoC. Even though this sounds like a reasonable idea, and the least time-consuming one, it can still generate unnecessary overhead. Namely, like most of the blocks in the system, the Boot ROM code also evolves and changes during the system's development. For example, some transactions in the power-up sequence need to be added, removed, or changed. Also, sometimes delays between transactions need to be changed, or the approach is changed (e.g. instead of delays, status register polling is used). This means that for every change, new information needs to be provided to the IP vendor, which then needs to regenerate the Boot ROM code and provide it back to the SoC owner. This would create a significant dependency on the IP vendor's availability. True, in case the Security block is developed in-house (and not licensed from a third-party IP vendor), or the SoC owner and IP vendor are very tightly coupled, this procedure might not be too complicated. Since this is not always the case, and usually the IP vendor has its own priorities and pace, this procedure could become cumbersome.

Another solution is to take the environment provided by the IP vendor and change the code for generating the Boot ROM to generate what is needed. This approach might require significant time and effort to understand the implementation and to be able to adapt it, which defies the purpose of obtaining the third-party IP in the first place.

Thus, the best solution is possibly the one in-between the first two mentioned options – the IP vendor creates place-holder functions in their own environment, which, once populated with power-up related code by SoC owner, can then be compiled and integrated into the Boot ROM data. In this case, the SoC owner needs to be familiar only with these place-holder functions, and to update them when the power-up sequence needs to be changed.

The Boot flow is different for Non-secure and Secure operation modes.

Non-secure boot flow

Non-secure boot flow is not intended to be used in real life scenarios but rather mostly used during the development and testing. On the other hand, Secure boot is not practical for usage in testing and development,

because once the Security IP is up and running in Secure mode, it locks all debug interfaces, making it almost impossible to debug issues that occur during testing.

Figure 2 presents a diagram of the Non-secure boot flow. This figure does not show the System Interconnect unit (Network-on-Chip), even though it exists between the blocks of the SoC (this was done to simplify the diagram). Also, NVM is an external memory, and is not part of the SoC as it seems from the diagram, but from the logical perspective it does not make any difference. The test *non_secure_boot_test* from Table 1 implements a complete scenario for the Non-secure boot. Before the run, the Boot ROM and OTP memories need to be preloaded with relevant data. Besides other information, OTP data also holds the information about the mode of operation – in this case Non-secure. Both Secure and Non-secure boot flows start the same way – they are triggered once the Security IP is out of reset. At this point, the Security IP starts reading the content of its own OTP memory. Since the Security IP is instructed to operate in Non-secure mode, it triggers the part of the Boot code allocated to that mode. The content of the Boot ROM memory is the same, regardless of the mode of operation. Different parts of the Boot ROM memory are being used, depending on the mode of operation. The Non-secure boot ROM flow performs internal configurations and generates previously mentioned external DMA transactions towards the rest of the SoC. One transaction is related to acknowledging the rest of the system that the Security IP is up and running. In case this transaction does not happen within a predefined time duration (i.e. if timeout occurs), the SoC considers the Security IP "dead", and performs a predefined sequence for that scenario. Another transaction generated by the Security IP is to power on the main CPU, which then starts executing its program. The execution of this program is also very important for this test, because, by executing it, we verify that the system is indeed powered on and that it can operate correctly. For example, after it is powered-on, the main CPU starts reading its IRAM and DRAM data from the NVM (which is preloaded at the beginning), and performs some Debug register write and read operations, with simple checking.

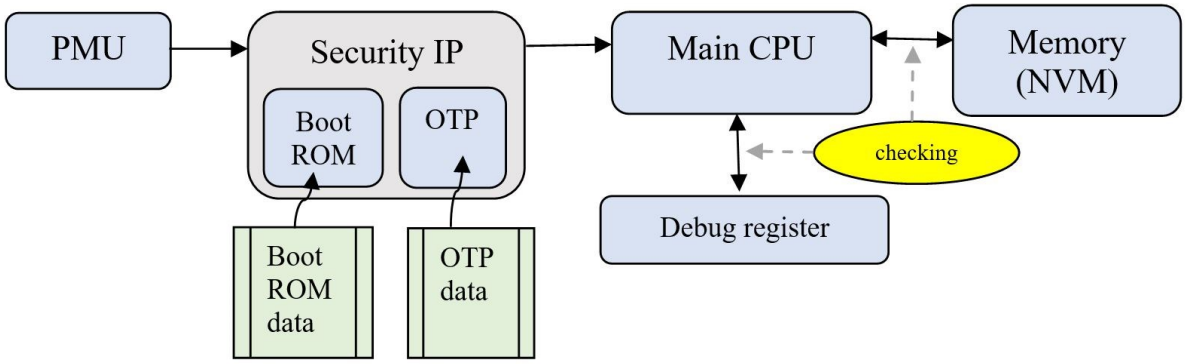

Figure 2. Non-secure boot flow – block diagram

Secure boot flow

Besides providing the Boot ROM and OTP code, some additional preparation needs to be done before the test run, in contrast to Non-secure boot. Namely, the NVM memory must be preloaded with the encrypted Allocation table (ALT) data, the Security IP's IRAM and DRAM data, and the main CPU's IRAM and DRAM data (we explain why later) – commonly referred to as the Encrypted Image. For that purpose, this encrypted data must be first created. Similar to the Boot ROM code, the SoC owner could provide all the necessary information to the IP provider, which could then generate the necessary database. Once again, this proved to be a cumbersome process in many cases, and the best solution is for the IP provider to create a placeholder configuration file, which is then populated by the SoC owner, to generate the encrypted database. This configuration file contains the information about the SoC's address space, crypto algorithm being used, confidentiality and authenticity modes of operation, and the path to the files holding the content of all necessary IRAM and DRAM memories, as well as the ALT file. The ALT file contains information required for handling the data, such as keys used for authentication. Once the file with the encrypted data is created, it usually needs to be further manipulated with specific scripts, to adapt the data format to the one suitable for preloading it to the memory model.

The test *secure_boot_test* from Table 1 implements a complete scenario for the Secure boot. Once all necessary data is preloaded, the Secure boot flow can start. Figure 3 describes a diagram of the Secure boot flow. The start of

the flow is similar to the Non-secure boot, but the difference is that the OTP is preloaded with information that holds the Secure mode of operation. Once the Security IP obtains this information, it immediately locks all interfaces and blocks within the SoC which are unnecessary for real life operation (these interfaces and blocks are mostly related to Debug features). Then the Security IP acknowledges to the rest of the system that it is up and running. Then, the Security IP starts reading the Encrypted Image from the NVM. First, the ALT data is read and decrypted. This is followed by the Security IP's IRAM and DRAM data, which is decrypted and stored inside the Security IP block. The step related to the main CPU's IRAM and DRAM data is the most important and the most critical from the perspective of IP integration verification, since it also generates many external transfers. The encrypted data is read from the NVM, decrypted inside the Security IP block, and then written through the DMA to the main CPU's IRAM and DRAM memories. Once this process is done, the Security IP configures the main CPU to read its own IRAM and DRAM by default, then powers on the main CPU, which then starts executing the program uploaded by Security IP. As with the Non-secure boot flow, this program's execution should also be checked - a few simple write and read transactions to the internal RW registers are sufficient, and we must confirm that data matches. Also, we must check that the Debug features are locked, by trying to access them and see that transactions failed.

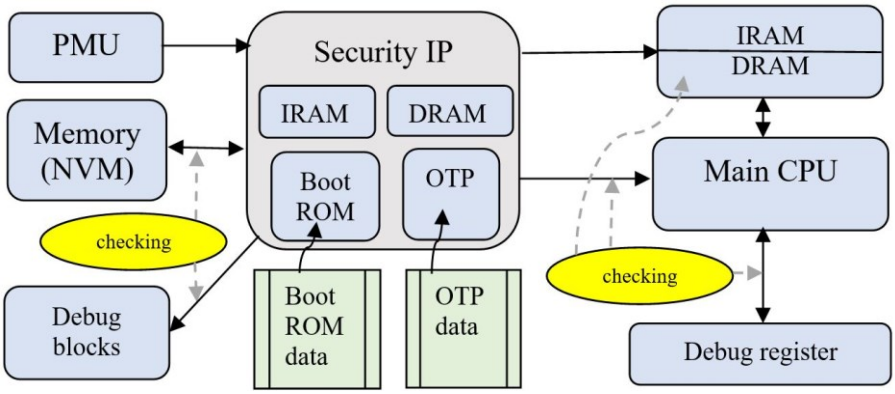

Figure 3. Secure boot flow – block diagram

The procedures described are very important for checking the correctness of the Security IP integration in the SoC, and they cannot be fully verified at the block level. This allows checking that the Security IP is properly connected to the PMU and the main CPU from the power-on perspective, whether it can successfully access the NVM, and most importantly, if the SoC owner is capable of creating its own specific Boot ROM and Encrypted Image databases.

Tests and debugging

To verify the Non-secure and Secure boot flows, with associated features and behavior, at least two separate tests are needed. However, since the Secure boot is more complex, and it locks Debug interfaces and blocks within the rest of the SoC, the best approach is probably to have a separate test for each Debug unit, or at least add them to the same test gradually.

When it comes to debugging, the Boot ROM code and Encrypted image proved to be the most difficult parts to handle. Even though it looks simple enough – just need to populate place holder functions and configuration file – once the Security IP is out of reset, very often one of the following two things happened: either nothing happened on the external interface, or an error occurred. This means that something is wrong with the Boot ROM code or the Encrypted image. Debugging this is very difficult for the SoC owner because the internal structure and implementation of the third-party Security IP is unknown. For such scenarios, the Boot Monitoring interface is available on the boundaries of the Security IP. This interface indicates when the boot flow started and ended, and also indicates if an error occurred including an error code. By checking the error code, the user can potentially understand what the issue is, to a certain extent. It is good practice to wire this interface all the way through to the boundaries of the SoC, so that the chip can be tested and debugged – however, this wiring needs to be verified as well. In most of the cases, the Boot ROM and the Encrypted image were corrupted because the placeholder

functions and configuration file were not compiled and handled correctly, and/or the file created with encrypted data was not manipulated correctly.

Checking, coverage, and RTL bugs

Since both tests end with the main CPU executing write and read transactions to registers, the most important task is to check that these transactions are performed correctly. If one of the steps that led to these transactions was incorrect, chances are that either these transactions would not be executed at all, or they will be executed incorrectly. However, in order to be sure that everything else went well, and to simplify the debugging process, other checkers can be placed on each of the steps in the flow: check if the Security IP executed its part of power-up sequence, check if all the necessary blocks were powered on, check the interface on the NVM to see if the data is being read correctly, and a backdoor check on the contents of the main CPU's IRAM and DRAM after they are populated by the Security IP. The simplest way to simulate suspicious behavior during the Boot flow is to intentionally corrupt the content of the memories - NVM, Boot ROM or OTP, and then check whether the Security IP detected and reported it.

As mentioned before, in the Secure boot mode of operation the Security IP locks all Debug interfaces and blocks within the SoC. It is essential to properly verify this, both from the connectivity and the functional perspectives. First, we need to check the Security IP's "lock" interface, to see if the correct "lock" signals were toggled. Then, we should test each Debug block itself to verify that the data does not propagate through.

Typical RTL bugs found in this process are listed in Table 2.

TABLE 2
RTL BUGS

Bug location	Description
PMU flow	Several bugs were found in this area. The order of transaction execution and timings were often wrong and needed to be adapted several times. Example of one specific bug: PMU sequence did not provide the clock and took main CPU out of reset on time - before Security IP tried to enable it. Therefore, this enable was never registered, and main CPU never started operating.
Security IP timeout	Duration of the Boot ROM flow was underestimated, therefore the timeout expired before the Boot flow was done, so the SoC considered Security IP "dead", and started the sequence that should be executed in that case.
AXI connectivity	ID port was internally made too narrow.
SPI debug port	Even when this debug port was locked by Security IP, it was still able to propagate the transactions inside, due to the bug in connectivity.

Since randomization does not play a big role in this flow, Coverage goals can be simple. The most important is to cover the activity on Debug blocks and interfaces once they are locked, to check that each interface is thoroughly stressed. Also, the Boot Monitoring and error reporting interfaces with external ports connected to them should be covered to verify that there was activity, especially when the error was intentionally introduced by changing the content of preloaded memories.

B. Communication between Security IP and the rest of the system, by initiating Security specific services

The Security IP can be operated by the main CPU by using a set of instructions, defined by the IP provider. The main CPU must be able to request services from the Security IP, like Crypto service and true-random number generation. To perform these services, the Security IP must be able to access memories and other blocks in the SoC, to obtain data and information. Therefore, it is essential for integration verification to generate tests that check such services.

TRNG generation (test name: *trng_test*) is the simpler of the two services mentioned, because it does not need to read data from the SoC, but rather only writes to it, and therefore is more convenient as a first test. Figure 4 presents a diagram of TRNG service flow. First, the set of instructions provided needs to be studied, and the sequence for the TRNG purpose should be created. Such a sequence is composed of service instructions, configuration, and external addressing. This sequence can be executed only after the boot flow is completed, so the Boot flow test passing is prerequisite. To ascertain whether or not the boot flow completed, polling on the status register must be done. Once the Security IP is ready, the instruction frame is sent and collected by the Mailbox, which is the block inside the

Security IP that is responsible for the handling of instructions. Once the Security CPU receives the instruction, it activates the TRNG block which generates a random number. This number is then transferred out through the DMA to the one of the memories or the main CPU of the SoC. The checking is done at the end, once the number is received. It is important to check if the number that exited the Security IP is the same as the one received in the memory. Coverage in this case can collect different configurations of the TRNG service, to verify that interfaces between the Security IP and the rest of the SoC can transfer any number created.

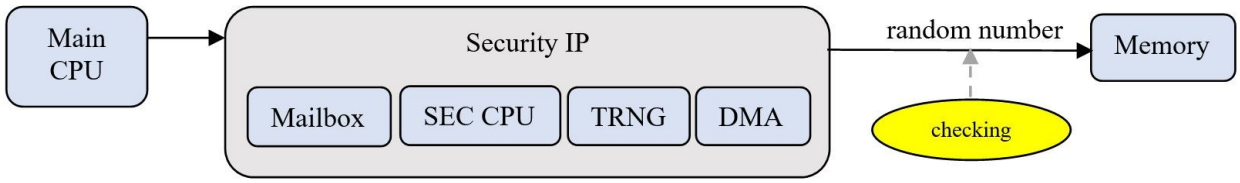

Figure 4. TRNG service flow – block diagram

Crypto service is more complex (test name: *crypto_service_test*), because the Security IP must read data from the SoC memories as well. Figure 5 presents a diagram of the Crypto service flow. The start is similar to the TRNG flow: polling is done to detect the end of the boot flow, and the sequence of instructions is sent. In this case, the size of data and information about the desired encryption algorithm must also be provided. Once the Security CPU receives the instructions through the Mailbox, it initiates the DMA data read. Data can be read from a few different memories – NVM, DDR or System RAM, and all of them need to be verified. For better performance, the Security IP is equipped with two DMA units, and can also work in single DMA mode - and both options must be verified. Once data is read from the memory, it is encrypted inside the Security IP unit, and written outside through the same DMAs. Checking is best done on-the-fly, once one transaction reaches its memory location, the data is compared to the data that left the Security IP. Coverage can also be done on the configuration, to make sure different amounts of data and Crypto algorithms were used.

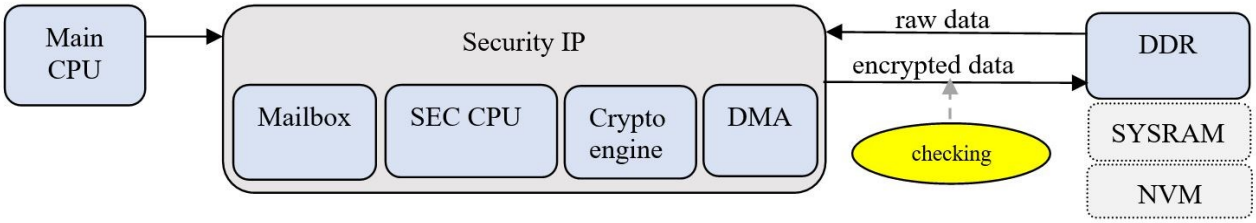

Figure 5. Crypto service flow – block diagram

C. Using the Debug interface for accessing the Security IP memory space

The Security IP contains two blocks dedicated to debugging: the Debug Mailbox and the Debug Interface. Figure 6 presents a diagram of the Debug Mailbox and Interface access flow. Through the Debug Mailbox, the Security CPU can receive instructions and perform operations similar to that of the main Mailbox. The Debug interface is used for accessing the Security IP internal Network-on-chip (NOC), through which the complete address space is available. This means that the content of most of the memories can be read, and more importantly, all internal status registers can be accessed as well, to obtain valuable information. These blocks are connected to JTAG, that is located on the external boundaries of the SoC. This means that in specific circumstances, the Security IP can be operated from the outside, and all necessary status information can also be obtained as well. This feature is very useful when the chip is being tested in the lab (e.g. if the Security IP is not responding upon power-up, its status can be easily checked).

On the other hand, this may represent a vulnerability, as it could also be potentially used by pirates to access the chip. However, specific steps are taken to prevent this scenario. First and foremost, these blocks are completely locked in Secure operation mode, and there is no option to unlock them. This means they are protected from being accessed in real life scenarios. They are also locked by default in Non-secure mode, but can be unlocked if the user performs a HW authentication, by transferring a specific sequence of instructions to the Debug Mailbox, followed by a specific authentication handshake that involves keys. If the HW authentication is successful, the Debug blocks

become accessible. It is essential to have dedicated tests to verify Debug blocks behavior in both Secure and Non-secure mode of operations. Test *debug_inf_non_secure_test* generates scenarios in the Non-secure mode of operation. Besides checking that the Debug blocks can be used in Non-Secure boot, it is also very important to check for unsuccessful HW authentication, to make sure that they do not get unlocked in that case as well. Test *debug_inf_secure_test* must verify that Security IP internal information cannot be reached through Debug interfaces in Secure mode of operation.

Randomization and Coverage in this case are very important. It is important to check that the complete address space of the Security IP can be accessed when Debug blocks are unlocked, and that none of the addresses can be accessed in Secure mode, and in unlocked Non-secure mode.

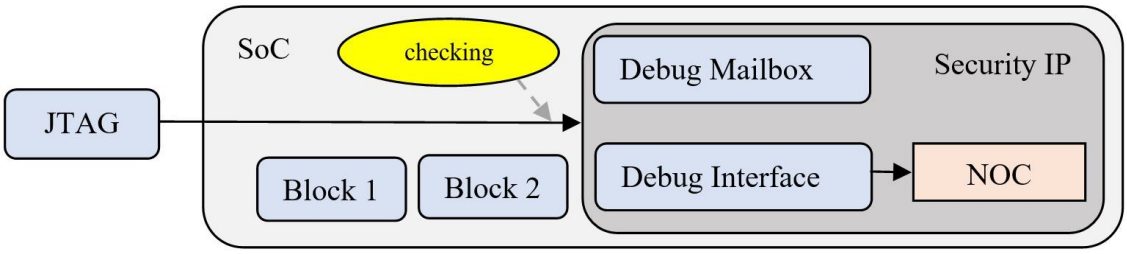

Figure 6. Debug Mailbox and Interface access flow – block diagram

D. HW alarm and Interrupts

The Security IP contains more than 50 Digital Sensors distributed over the block area, with the purpose of protecting against Fault-Injection-Attacks. Figure 7 shows a diagram of the HW Alarm and Interrupts access to the controller. Attacks can be performed by injecting faults through input clock frequency and/or input voltage, increasing temperature and radiation. Test *hw_alarm_interrupts_test* should contain scenarios for setting both alarm and interrupts. From the integration perspective, it is sufficient to make the alarm trigger in any possible way, and check whether it is correctly connected to the rest of the SoC. This means that the test scenario can be created by generating overclocking, then checking that the alarm is triggered, and then making sure it propagates to the Interrupt controller in the SoC, which then triggers the necessary actions.

A similar approach applies to interrupts as well. The Security IP block can generate several different output interrupts, mostly related to an ECC error, Authentication error, OTP error, etc. All interrupts are already functionally verified at the Block level, so at the integration stage only their connectivity to the Interrupt controller and related actions must be checked. Coverage is simple, just check that all interrupts occurred at the Interrupt controller.

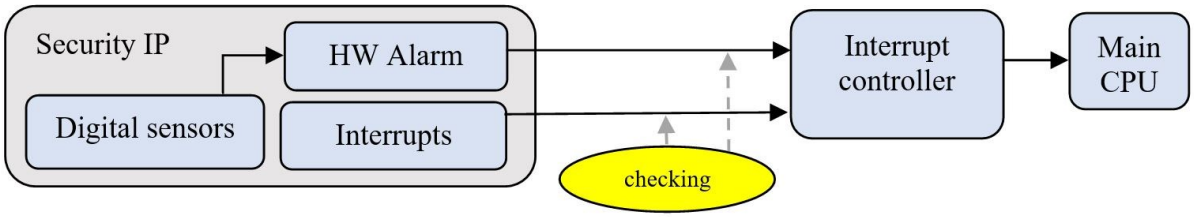

Figure 7. HW Alarm and Interrupts access to the controller – block diagram

E. Performance preservation

Performance is one of the essential topics that needs to be checked in integration, because the SoC Pipe throughput must be verified. At the Block level, only the interface throughput can be checked. Performance must be agreed and defined in advance, and, when it comes to verification, the number of tests must match the number of all possible pipes and configuration options that can affect the size of data and data processing speeds. For example,

some encryption algorithms generate more data and have slower calculation time than others. Also, the test must contain realistic scenarios, and not just a simplified one. For example, the simplified scenario would be if the Security IP reads data alone from the memory, encrypts it, and then writes it back. This scenario is oversimplified because, in reality, in addition to the Security IP, some other blocks will access the same memory at the same time, which could have a significant effect on performance.

The test with the most critical path and scenario will be described – *performance_path_1_test*. Figure 8 presents a diagram describing a critical scenario where the performance was checked. The Security IP needs to perform a Crypto service, i.e., reading the data from the DDR memory, encrypting it, and then writing it back. In parallel, the Ethernet unit writes the raw data into the same memory. This Security IP contains two DMA units, which must be used in parallel in order to achieve the desired performance. This fact complicates both RTL implementation and Verification tests, but has a significant positive impact on performance, so the complexity is justified. Therefore, performance bugs are likely to occur on the data path, most likely in NOC. In this case, a bug was discovered – the NOC did not have a buffer that was large enough to store the data from the largest burst that can be generated by the Security IP, so unexpected backpressure was generated, which had a negative effect on performance.

When it comes to checking, standalone performance checkers connected to the Security IP interface should be used. These checkers need to take a few configuration options, such as the expected throughput, and the triggering events to start and stop measuring. Once the measuring is done, the expected and calculated numbers need to be compared.

Coverage is also important for performance. Coverage needs to confirm that the desired scenarios are indeed created. This can be done through collecting the Crypto service configuration provided to the Security IP, as well as collecting configurations used in other Blocks involved in the throughput measurement.

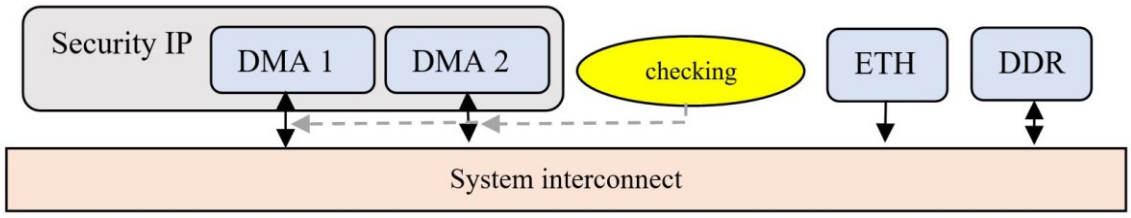

Figure 8. Performance checking - block diagram of critical scenario

Verification of Security IP integration contains some additional tests, such as OTP and MBIST, but there are no Security specific aspects about them, so they are not in the scope of this paper.

IV. CONCLUSIONS

Verification of Security IP integration represents an increasingly time-consuming and resource-demanding task because the Security blocks become more complex with each iteration. This paper described a relatively simple and compact, yet effective, method to verify correct integration of a Security IP into a SoC, by using tools and methodologies available to most semiconductor companies. We provided a description of common gaps between Block level and System level verification, presented a minimal set of tests, and a step-by-step guide on how to implement the solution. In addition, we also listed all the RTL bugs that were found during the process and exposed some critical points that needed special attention.

REFERENCES

[1] ISO/SAE 21434:2021 Road vehicles - Cybersecurity engineering
[2] Anders Nordstrom, Jagadish Nayak, “Building a Comprehensive Hardware Security Methodology“, DVCon US 2022
[3] Brent Sherman, Mike Borza, Brian Rosenberg, Charles Qi, “Security Assurance Guidance for Third-Party IP” J Hardw Syst Secur (2017)
[4] Sohrab Aftabjahani, An Overview of Security Annotation for Electronic Design Integration (SA-EDI) Standard, DVCon US 2022
[5] Brent Sherman, Mike Borza, et al. Security Annotation for Electronic Design Integration Standard (2021)
[6] Subin Thykkoottathil, Nagesh Ranganath, “Making Security Verification “SECURE”” DVCon US 2018
[7] IEEE 1800.2-2020 Standard for Universal Verification Methodology