Computers, Materials & Continua
DOI: 10.32604/cmc.2022.029794
Article

Tech Science Press

An AOP-Based Security Verification Environment for KECCAK Hash Algorithm

Hassen Mestiri[1,2,3,*], Imen Barraj[1,4,5] and Mohsen Machhout[3]

[1]Department of Computer Engineering, College of Computer Engineering and Sciences, Prince Sattam bin Abdulaziz University, Al-Kharj 11942, Saudi Arabia
[2]Higher Institute of Applied Sciences and Technology of Sousse, University of Sousse, Tunisia
[3]Electronics and Micro-Electronics Laboratory, Faculty of Sciences of Monastir, University of Monastir, Tunisia
[4]Electrical Engineering Department, Systems Integration & Emerging Energies (SI2E), National Engineers School of Sfax, University of Sfax, Tunisia
[5]Higher Institute of Computer Science and Multimedia of Gabes, University of Gabes, Tunisia
*Corresponding Author: Hassen Mestiri. Email: h.mestiri@psau.edu.sa
Received: 11 March 2022; Accepted: 20 April 2022

Abstract: Robustness of the electronic cryptographic devices against fault injection attacks is a great concern to ensure security. Due to significant resource constraints, these devices are limited in their capabilities. The increasing complexity of cryptographic devices necessitates the development of a fast simulation environment capable of performing security tests against fault injection attacks. SystemC is a good choice for Electronic System Level (ESL) modeling since it enables models to run at a faster rate. To enable fault injection and detection inside a SystemC cryptographic model, however, the model's source code must be updated. Without altering the source code, Aspect-Oriented Programming (AOP) may be used to evaluate the robustness of cryptographic models. This might replace conventional cryptanalysis methods in the real world. At the ESL, we discuss a unique technique for simulating security fault attacks on cryptographic systems. The current study presents a fault injection/detection environment for assessing the KECCAK SystemC model's resistance against fault injection attacks. The approach of injecting faults into KECCAK SystemC model is accomplished via the use of weaving faults in AspectC++ based on AOP programming language. We confirm our technique by applying it to two scenarios using a SystemC KECCAK hash algorithm case study: The first concerns discuss the effect of the AOP on fault detection capabilities, while the second concerns discuss the effect of the AOP on simulation time and executable file size. The simulation results demonstrate that this technique is fully capable of evaluating the fault injection resistance of a KECCAK design. They demonstrate that AOP has a negligible effect on simulation time and executable file size.

Keywords: Aspect-Oriented Programming; SystemC; AspectC++; cryptographic circuits; KECCAK; fault detection; fault attacks

1 Introduction

Electronic cryptography devices are commonly employed to safeguard sensitive data in embedded systems. These devices securely hold the secret key that is required to run the cryptographic algorithm. Algorithms are constructed and examined in order to guarantee that they are resistant to mathematical attacks. However, when implemented on hardware devices, the method becomes subject to physical attacks. The objectives of attackers of cryptographic equipment are self-evident. They want to get access to this secret key, private information, or even to disrupt the usual execution process. Fault injection attacks are one of the most effective forms of attacks against electronic cryptographic systems. They work by inserting faults into the algorithm's structure in order to retrieve the secret information.

Nowadays, the difficulty of designing cryptographic embedded systems outpaces developers' design and verification capabilities. A standard language for verifying and modeling complex systems, SystemC [1], was developed to address these problems. For modelling the insertion of faults into hardware platforms and System on Chip (SoC) architectures, it was regarded an excellent modeling language. There have been numerous approaches to simulating cryptographic algorithm security fault attacks [2–5]. In order to inject and identify faults, the majority of these methods necessitate changes in SystemC code by developers. Aspect-Oriented Programming (AOP) is a major way to avoid modifying the cryptographic design source code under test [6].

We developed a methodology, in this article, for simulating fault attacks security on KECCAK SystemC models at ESL level. For hardware design, we utilized SystemC as the system-level modeling language and AspectC++ as the AOP language [7]. We develop an AOP-based system-level fault injection/detection environment using SystemC and AspectC++ in order to test the KECCAK design's resilience against fault injection attacks.

This paper is organized as follows. The second section discusses the background knowledge, while the third section discusses the related work. Section 4 discusses the proposed SystemC-AOP approach for simulating fault attacks security of KECCAK systems. The proposed methodology will be evaluated in Section 5 through simulation results Analysis. Section 6 concludes the paper.

2 Background

This section discusses the required background material and terminology for the remainder of the wok.

2.1 Aspect-Oriented Programming

The AOP paradigm is a reasonably recent one that adds modularity to object-oriented languages such as SystemC. It enables the separation of crosscutting issues into distinct modules. These difficulties often originate as a functional unit but become diffused or entangled within the system. AOP enables the encapsulation of a common concern in a single unit of code called aspect code. An aspect weaver is used to insert the aspect code at compile-time or run-time.

Numerous significant concepts are included in an aspect-oriented language: [6]

- Advice: provide a computation that may be done after, around or before a specific join point.
- Aspect: address crosscutting concerns in a modular fashion and are a C++ extension to the class notion. Aspects may also have advice declarations in addition to attributes and methods.

- Join points: These are locations inside the component's code where two or more aspects may interact. A join point is a method, a property, a type, an object, or a point of access to a join point.
- A pointcut: expression specifies a collection of join points. Pointcut expressions are constructed using match expressions for the purpose of locating a collection of join points.

To incorporate new functionality as an aspect into an existing system, connect points within the core code must be identified to indicate where the aspect should operate. The aspect code is introduced at compile-time or run-time using an aspect weaver tool. The aspect weaver weaves aspect code into component code as seen in Fig. 1.

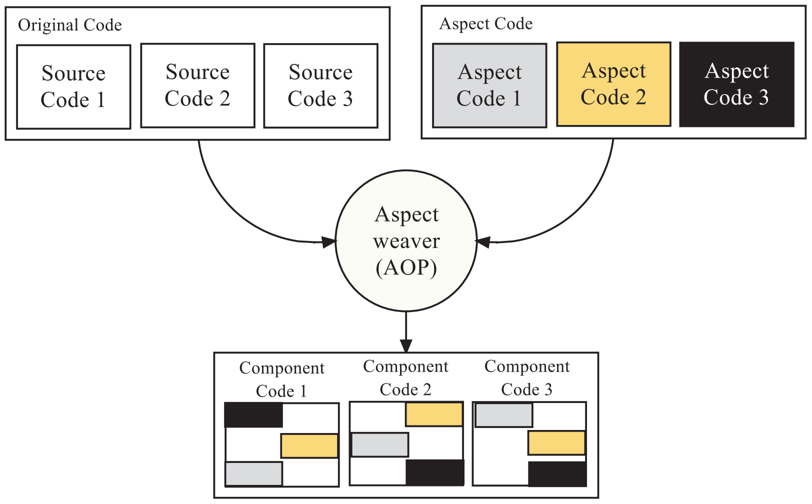

Figure 1: Weaving aspect module into original code

2.2 *KECCAK Algorithm*

The Keccak algorithm is defined by the permutation f. This function is widely used to represent a fixed-length KECCAK state with a length of $b = c + r$ bits, where c and r denote capacity and bit rate, respectively. In direct proportion to the bit-rate r values, KECCAK data speeds increased, and in direct proportion to the capacity c values, its security level improved. A new message with a length equal to or greater than the length of the KECCAK input message is created. Following that, the absorption phase is carried out in a five-step fashion for each KECCAK round. Finally, while the state initial r bits feedback as the data output block, the squeezing phase is done [8].

In this paper, we will concentrate on the proposed KECCAK type: The bits-width of the underlying permutation is Keccak-f [1600], where c equals 1024, r equals 576 and f equals 1600. The Keccak state is composed of a 55-lane array with a data length of w equals 64 bits. Each cycle of KECCAK is 24 rounds long and executes the five operations iota, chi, pi, rho and theta. These operations are state bits' permutations and simple logical manipulations.

2.3 *Fault Attacks*

Fault injection attacks are a kind of physical attack in which faults are introduced into the algorithm's structure in order to recover the KECCAK sensitive data. The cryptanalyst injects flaws into the processing algorithm as it runs. This causes an interruption in the normal execution process, resulting in the creation of incorrect ciphertext. Thus, after a certain number of fault injections and examination of incorrect ciphertexts, the cryptanalyst may recover the encrypted information [9–11].

3 Related Work

3.1 AOP for Software

AOP is becoming an essential paradigm for testing system applications [6]. In [12,13] java program has been instrumented with AspectJ weavers [14]. AspectC++ [7] which is a C++ based AOP weaver is more and more applied to software validation [15,16]. Etienne et al. used C++ aspects to identify bugs due to memory leaks, incorrect algorithm implementation, or interference among threads [17]. Aspects are automatically generated and weaved in C++ programs to dynamically report software bugs. Similar work that concern the testing of embedded Operating System (OS) is presented in [18]. They used manual aspects to test embedded C++ programs in OS. They highlighted four areas of functional testing which are memory, performance, robustness and coverage of C++ programs [18]. In [19,20] aspects are instead implemented for kernel testing. Interrupt synchronization have been well described for operating systems.

Through the use of an existing Ada aspect extension, Ada Aspects were applied to a real-time operating system in order to adjust its implementation to meet real-time restrictions [21]. They then determine its limitations and provide strategies for adapting it to real-time restrictions. Finally, they suggest a new compiler/weaver architecture to address the shortcomings of the old one.

Software security hardening using AOP is the focus of [22]. Secure patterns have been elaborated in AOP and applied to secure applications by the encryption of memory code. Similar approach is presented in [23]. AOP is a technique for introducing fault tolerance into distributed embedded system applications. The authors take into account a variety of fault-tolerant methods and redundant hardware/software setups. AOP is used to provide fault tolerance in the system at the application thread level. The benefits of this AOP-based strategy include increased modularity, reduced effort for legacy system modernization, and improved configuration for testing and product line development.

The analysis of relevant studies on AOP-based software demonstrates that there is great interest in applying aspects to program functional testing. However, few approaches concentrate on AOP oriented to security software hardening and fault-tolerant systems.

3.2 AOP for Mixed Hardware/Software Attacks

AOP specialization in literature is also applied in the scope of mixed hardware/software platforms for exploring SoC architectures. Using AspectC++, an Electronic Design Automation (EDA) tool is proposed to produce unified description of hardware and software components in [24]. Hardware and software characteristics are weaved into aspect programs and linked to the unified description of SoCs. The target architecture is a mixed Field Programmable Gate Array (FPGA)-based hardware/software implementation [24]. SoC exploration was also proposed in [25]. In [26], a new language called LARA with the appropriate toolchain is developed to instrument application codes with specific features like monitoring, logging and debugging. Hardware/software specialization such as functional and non-functional requirements are weaved into C++ aspects to improve code modularity. The AOP layer in LARA guide designers to implement optimized FPGA-based hardware/software applications.

The analysis of mixed hardware/software specialization has showed that AOP is used to separate hardware concerns from software ones for exploring SoC architecture. However, the applications of AOP to model and/or implement extra specialization of encrypted data are not considered.

3.3 AOP for SystemC Modeling and Synthesis

With pure hardware implementations AOP, is mainly the focus of functional verification. Few approaches have applied AOP for designing or modeling hardware components [7,27]. In these works, aspects provide explicit architecture notions, such as concurrency and time, resulting into synthesizable descriptions of a SoC. With SystemC modeling, AOP approaches extract SoC aspects like performance measure, communication and cache policies [28]. Resulting SystemC models are difficult to synthesize, so such approaches are rather more useful in the area of verification and simulation like in [29].

AOP applied with the SystemC synthesizable subset is proposed in [30] where aspects are used to schedule the model execution dependencies. Designing electronic circuits using aspects is not a good practice according to our point of view. This requires to deviate AOP programming languages for providing explicit hardware notions; and may result into unoptimized circuit model compared to dedicated hardware languages like Very high speed integrated circuit Hardware Description Language (VHDL) or Verilog.

4 Proposed KECCAK SystemC-AOP Environment

4.1 Proposed Approach: General View

In the proposed approach, we used the AspectC++ as AOP programming language and SystemC as modeling language at ESL for hardware design. Using AspectC++ and SystemC jointly at system-level, we design a KECCAK AOP environment to test the robustness and the detection capability against the fault attacks. Our proposed KECCAK SystemC-AOP environment was designed to reach the following objectives:

- Simulate the KECCAK security fault attack at ESL,
- Check the robustness and the detection capability of KECCAK SystemC/AOP model against the fault attacks,
- Control the fault injection/detection process: fault time and fault location,
- Inject and detect the faults at real time process.

Fig. 2 presents a global view of our approach to analysis the KECCAK model robustness against the fault attacks.

As seen in Fig. 2, the KECCAK SystemC-AOP flow consists of four main phases. The Aspect modeling phase has the aspect modules and the KECCAK SystemC. The aspect module consists of three aspects: KECCAK Fault Injector (KFI), KECCAK Fault controller (KFC) and KECCAK Fault Analysis (KFA). The KFI inject the faults at the exact time and location. Driving the synchronization between the KECCAK SystemC/AOP modules are performed via the state controller KFC. The KFA generates fault attack report contains information about the fault detection, the fault classifications, as well as the injected faults and their effect on the functional designs.

The second phase is the Aspect weaving phase, it consists of 2 steps: the aspect weaving and the model compilation. The three aspect codes presented in the first phase are described using AspectC++, so the weaving of aspect codes into the KECCAK SystemC design are performed without any SystemC code modification. As soon as the weaving process is performed, the obtained source code is compiled and an executable file is generated.

The third phase in the KECCAK SystemC-AOP environment is the Aspect simulation phase which consists of two modules: the executable file system and the reference KECCAK model. The executable file system is the weaving outcome of the aspect modules into the KECCAK SystemC design. The reference KECCAK model presents fault free KECCAK model. As shown in Fig. 2, the outputs of the two modules are checked in order to find any occurred error in the executable file system.

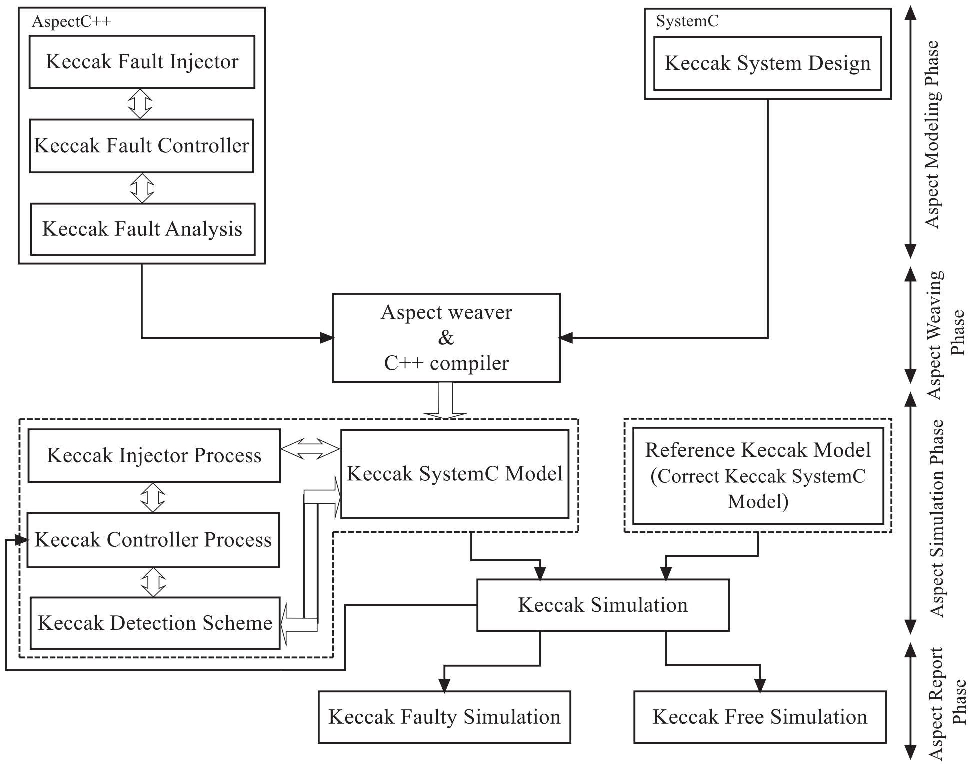

Figure 2: Proposed robustness evaluation flow: General view

The last KECCAK SystemC-AOP environment is the Aspect report phase which contains information about the fault impact on the KECCAK design, as well as the classifications of the injected faults: Detected Error, Undetected Error, Silent Fault and False Positive.

4.2 KECCAK Injection and Detection Using AOP

4.2.1 KECCAK Fault Controller

The KFC module is interfaced with the KFI and KFA modules to determinate all injection and detection tasks. The KFC process is based on a Finite State Machine (FSM) to ensure an efficient injection and detection fault controllability with minimum perturbation to the target system. Fig. 3 illustrate the proposed FSM for KFC.

The different KFC states are described using the FSM. The KECCAK fault attacks state processes if the *Start_fault_attack* transition is set. Then the KFC module transmits to the KFI module the next state depending on the selected fault type: Permanent or Transient. Then, if the *Transient fault attacks* state is executed, two states are generated: *Transient single bit* and *Transient multiple bit*. Whatever the state executed, the KFC module provides to the KFA module the required information to generate the *Fault attack report*. The KECCAK fault detection state executes if the *Start_fault_detection* transition is set. Then, the *Fault detection report*, which included the fault detection information, is generated.

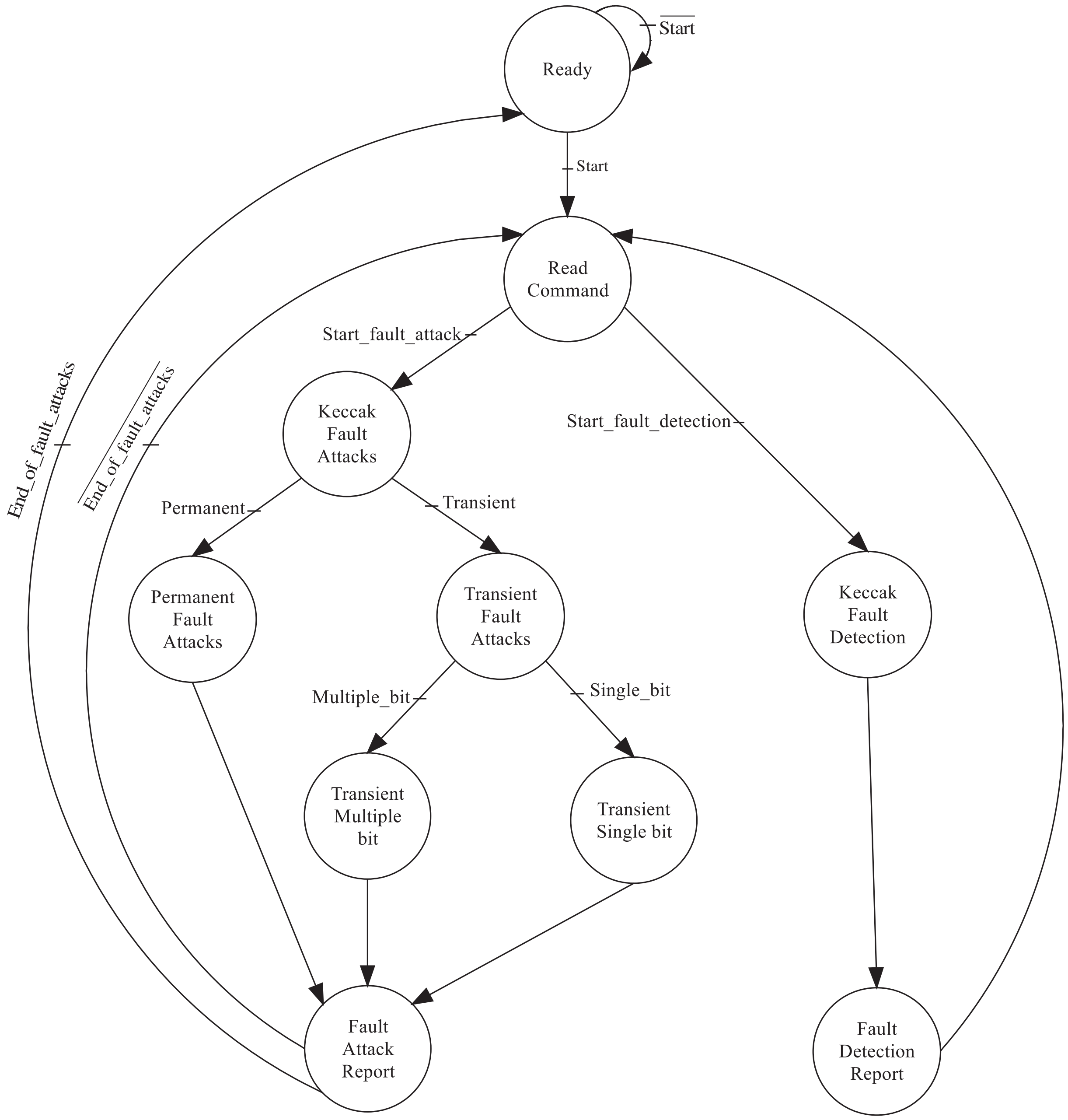

Figure 3: KFC finite state machine

Lines 2, 3, and 4 of KFC module describe the command, attacks and detection pointcuts, respectively: in the keccak class, all executable process named *process_1*, *process_2*, and *process_3*. On line 11, the advise *advice (execution(attack())): after ()* was declared, which implies that when the *attacks()* function is used, the code in this advise is performed after the code specified by that function in the SystemC module. Listing 1 describe the pseudo code of the aspect used by KFC module.

Listing 1: Pseudo code of aspect KECCAK_Fault_Controller

```
Aspect KECCAK_Fault_Controller {
     pointcut command() = “%keccak::process_1(...)”;
     pointcut attack() = “%keccak::process_2(...)”;
     pointcut detection() = “%keccak::process_3(...)”;

     advice (execution(command())): after () {
          if start
              read_command();
             ...
      }
      advice (execution(attack())): after () {
           if start_fault_attack {
              keccak_fault_attack ();
              if transient
                 transient_fault_attacks();
                 if single_bit
                     transient_single_bit();
                  else if multiple_bit
                     transient_multiple_bit();
             else if permanent
                  permanent_fault_attacks ();
             fault_attack_report ();
           }
           ...
      }
      advice (execution(detection())): after () {
           if start_fault_detection {
             keccak_fault_detection ();
             fault_detection_report ();
           }
           ...
      }
      ...
   }
```

4.2.2 *KECCAK Fault Injector*

The KFI module is interfaced with the KFC module to allow the fault injection which stocked in the fault database. The KFI module can generates three fault types: permanent fault, transient single fault and transient multiple fault. The functionality principle of the KFI as presented by the flowchart in Fig. 4.

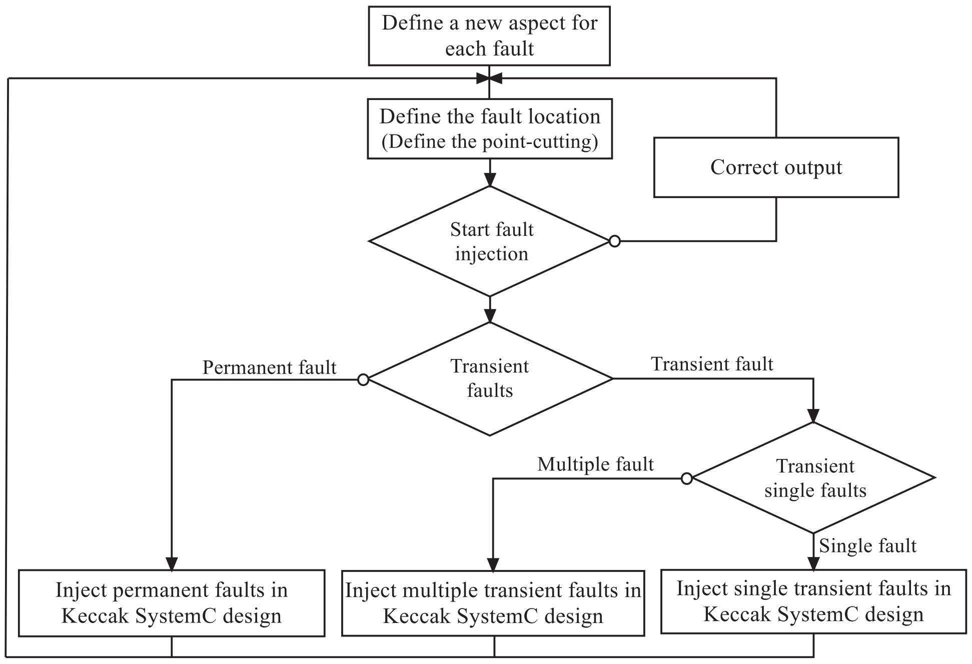
Define a new aspect for each fault
Define the fault location (Define the point-cutting)
Correct output
Start fault injection
Transient faults
Permanent fault
Transient fault
Transient single faults
Multiple fault
Single fault
Inject permanent faults in Keccak SystemC design
Inject multiple transient faults in Keccak SystemC design
Inject single transient faults in Keccak SystemC design

Figure 4: Aspect KECCAK fault injector flowchart

The KECCAK SystemC-AOP environment generates ports to allow the KFI reading and modifying the KECCAK state in all possible locations. A typical ports injector cryptographic model can be described in Fig. 5 which be executed to evaluate the KECCAK SystemC-AOP design via three locations: Fault Injector Interconnections (FII), Fault Injector Bus (FIB) and Fault Injector Register (FIR). The typical ports injector cryptographic model is consisted of five modules:

- The input and output buffer data,
- KECCAK module used to hash the input message,
- Control module used to realize the synchronization between all modules,
- KECCAK bus is used to communicate between all modules.

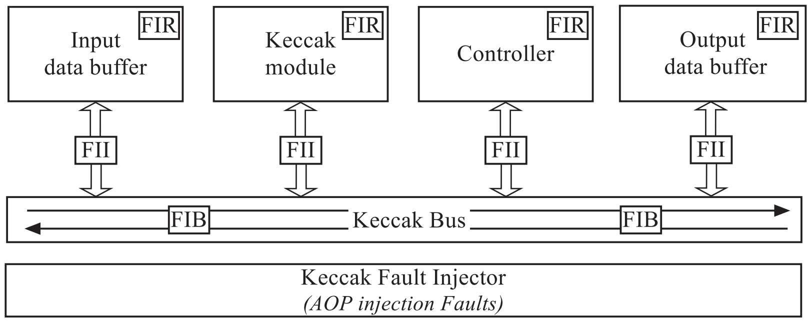

Figure 5: KECCAK fault injection locations

As presented in Fig. 5, the fault can be injected in all possible locations as memories, registers, peripherals and the interconnections of functional modules which means all possible faults can be modeled and the fault location is not limited to storage modules but includes the communication modules as the bus data. The fault injection process is executed using AOP approach which means no KECCAK code modification is needed.

The uses of FII to create faults into the interconnections between all modules does not need any block design modification since the fault injection process is performed using AOP. Using this process, the interconnection data will be modified inside the FII. This is reached using saboteur which consists to apply a mask for the data communicated inside the FII. Listing 2 shows the pseudo code of the aspect used by saboteur.

Listing 2: Pseudo code of aspect Saboteur

```
Aspect Saboteur {
     pointcut FII() = "%KECCAK_SHA3::KECCAK(...)";

     advice (execution(FII ())): after () {
          if start_fault_injection {
             faulty_output = apply_mask(in.read());
             data_out.write(faulty_output);
          }
          else
              data_out.write(in.read());
              ...
      }
      ...
}
```

The FIB are inserted in the decoder, data FIFO and bus design to inject the fault into the transmitted data on bus. The data are modified inside the FIB. The uses of FIR to generate fault in the KECCAK registers allow not only the signal modification, but the access to modify the state variables.

The FII, FIB and FIR are connected to the KECCAK SystemC-AOP environment to control the fault types to be injected as well as the locations and the times of the attacks against the KECCAK model.

4.2.3 KECCAK Fault Analysis

The KFA module is developed to check the correct operation of the functional designs either against fault attacks or free of attacks. The functionality principle of the KFA as presented by the flowchart in Fig. 6.

A fault detection scheme is necessary to secure the secret KECCAK information from cryptanalyst that may maliciously inject faults. The communicated message in the faulty and the correct SystemC models are checked using the checker module to detect the occurred or injected faults occurring at run time. An analysis report is created at end of detection process which contains detection attack results and the effects of the faults on the functional KECCAK designs.

The aspect *KECCAK_Fault_Analysis* starts if the *Start_fault_detection* is set. The *KECCAK_fault_detection* function uses *data_in* as input and returns the variable *result* as output. The content of the *result* is print in the *Fault_detection_report*. Listing 3 describe the pseudo code of the aspect used by KFA module.

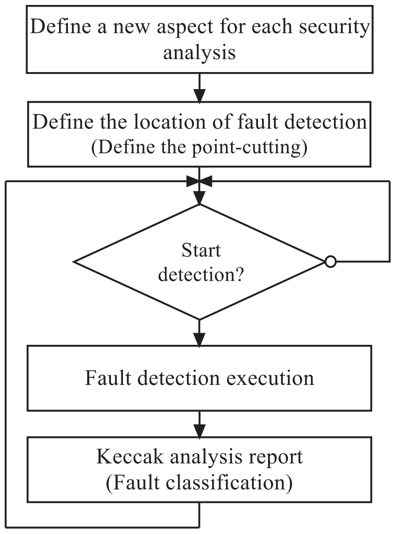
Define a new aspect for each security analysis
Define the location of fault detection (Define the point-cutting)
Start detection?
Fault detection execution
Keccak analysis report (Fault classification)

Figure 6: Aspect KECCAK fault analysis flowchart

Listing 3: Pseudo code of aspect KECCAK_Fault_Analysis

```
Aspect KECCAK_Fault_Analysis {
     pointcut detect_loc() = "%KECCAK_design::KECCAK(...)";
     advice (execution(detect_loc())): after () {
          if start_fault_detection {
              result = KECCAK_fault_detection(data_in.read());
              Fault_detection_report(result.read());
          }
          ...
     }
     ...
}
```

5 SystemC-AOP KECCAK Environment Validation

This section is dedicated to evaluate the proposed KECCAK SystemC-AOP environment. As a cryptographic model, we used the KECCAK hash function. To begin, we ran a series of fault injection simulations to assess the SystemC-AOP verification environment's ability to detect faults. Second, we examine the effect of the AOP on simulation time and executable file size. Thirdly, we analyse the effect of the AOP on design process. SystemC 2.3.2 and AspectC++ 2.2 were used to describe all designs. All simulations were run on an Intel Core I3-4010U 1.7 GHz processor with 6 GB RAM and gcc version 7.2.0.

5.1 Impact of AOP on Fault Detection Capabilities

To validate the proposed KECCAK SystemC-AOP environment's efficiency in evaluating KECCAK's robustness against fault attacks, we perform fault attacks using the proposed environment to assess the KECCAK SystemC-AOP detection capabilities. The capability for fault detection is determined using two scenarios: pure SystemC and SystemC-AOP, as well as the fault detection schemes [8]. To do this, we leverage the SystemC and AspectC++ simulation kernels to validate and compare the KECCAK SystemC-AOP environment results.

The SystemC scenario entails the usage of SystemC as a system-level modeling language, which necessitates a change to enable fault attacks and detection throughout the KECCAK process. AspectC++ and SystemC are combined in the SystemC-AOP scenario used in the KECCAK SystemC-AOP environment. As a result, this scenario does not require any modification to the model's SystemC KECCAK code. Transient and permanent faults are included in all scenarios.

5.1.1 The Impact of AOP on Single-Fault Detection Capabilities

We began by examining the KECCAK cryptographic model's detection capabilities for faults impacting a single bit. The transient single-bit fault is injected affects 1 bit where the simulation security is executed using 1,000,000 faults. Faults have been inserted meticulously into every byte, operation, and KECCAK round.

As seen in Tab. 1, the simulation results indicate that both scenarios identified all inserted transient single-bit fault in the KECCAK SystemC models. This means that the fault detection capabilities of pure SystemC and mixed SystemC-AOP cryptographic models are equivalent, and hence that our SystemC-AOP technique is valid.

Table 1: Detection capability of transient and permanent faults

Type of faults	Fault multiplicity		Percentage of the undetected faults		Percentage of the false positive		Percentage of the detected faults	
			SystemC	SystemC + AOP	SystemC	SystemC + AOP	SystemC	SystemC + AOP
Transient faults	Single-bit		0.542	0.542	6.356	6.356	93.102	93.102
	Multiple-bit	N = 2	0.256	0.256	4.984	4.984	94.760	94.760
		N = 3	0.087	0.087	2.521	2.521	97.392	97.392
		N = 4	0.036	0.036	1.382	1.382	98.582	98.582
		N = 5	0.015	0.015	0.783	0.783	99.202	99.202
		N = 6	0.011	0.011	0.403	0.403	99.586	99.586
		N = 7	0.008	0.008	0.252	0.252	99.740	99.740
		N = 8	0.005	0.005	0.173	0.173	99.822	99.822
		N = 9	0.002	0.002	0.128	0.128	99.870	99.870
		N = 10	0.001	0.001	0.092	0.092	99.907	99.907
		Random	0.003	0.003	0.074	0.074	99.923	99.923
Permanent faults	Single-bit		0.158	0.158	5.953	5.953	93.889	93.889
	Random-bit		0.015	0.015	0.084	0.084	99.901	99.901

5.1.2 The Impact of AOP on Multiple-Fault Detection Capabilities

In the second experiment, we examined the KECCAK SystemC model's detection capabilities for faults affecting at least two bits. 9 tests, each with a different fault multiplicity, replicate the KECCAK SystemC model implementation. This model has been evaluated using 1,000,000 faults. N is the number of faulty bits in each block data. Faults have been inserted meticulously into every round, operation and byte.

Random transient and permanent faults are introduced into any of KECCAK states, with the number of erroneous bits ranging from 2 to 10. Overall, 11 tests are used to replicate the KECCAK model, each one identified by the bit number of the inserted faults.

The proportion of undiscovered multiple faults for the protected cryptographic model utilizing two situations is shown in Tab. 1. By comparing the results, we can observe that the KECCAK SystemC model's fault detection capabilities, for transient and permanent faults, are always equivalent whether the two situations are used. This demonstrates that our SystemC-AOP technique is effective.

5.2 The Impact of ESL on the Simulation Time and Executable File Size

To examine the influence of the AOP on simulation time, we placed 2,000 random faults into the cryptographic model's likely fault locations and measured the mean simulation time. Prior to presenting and analyzing the data, it is vital to note that the AOP strategy's influence on simulation time is proportional to the number of join point data given to the original SystemC code by the advice code.

The user time (uTime) and kernel time (kTime) simulations for the two situations are shown in Tab. 2. Indeed, the difference is less than the margin of error associated with the measuring method we used (the Linux command time). Our conclusion is that the AOP has a negligible effect on the simulation duration and is not a factor that should exclude its usage in cryptographic verification security. Similarly, we determined the executable files size of SystemC and SystemC-AOP created by AspectC++ and the SystemC kernel, respectively (see Tab. 2). We demonstrate that AOP has no discernible effect on the executable file size.

Table 2: Simulation of time analysis and file executable size with and without AOP

Simulation type	Simulation time				Executable file	
	kTime (s)		uTime (s)			
Scenarios	SystemC	SystemC + AOP	SystemC	SystemC + AOP	SystemC	SystemC + AOP
Protected KECCAK design [8]	0.045	0.043	2.645	2.649	1.598 MB	1.602 MB
Protected KECCAK design [10]	0.049	0.048	2.841	2.838	1.612 MB	1.615 MB

5.3 The Impact of ESL on the Design Process and Simulation Time

The following studies were done to assess the ESL's attention span and simulation length throughout the design process. Transaction Level Modeling (TLM) with Programmer View Timed (TLM-PVT) is the ESL modeling level, and SystemC is the modeling language. At the Register Transfer Level (RTL), the same KECCAK model is modeled using VHDL. Both VHDL and SystemC cryptographic models are simulated using the identical conditions.

The coded lines number was determined for both description levels. It is discovered that the RTL code is four times as lengthy as the ESL code. Additionally, we compared the week-by-week development cycle. RTL coding took 19 weeks, whereas ESL modeling took just 4 weeks. This eliminates about 79% of the time spent on design. Additionally, we examined the time required to simulate a security attack in both scenarios using 1,000,000 random faults introduced in various places. In both situations, we covered over 99% of the conceivable injection places. The simulation findings indicate that the ESL significantly accelerates and decreases the time required to simulate a security attack by a factor of 30. The encrypted ESL environment proposed here is significantly more straightforward and easier to deploy than RTL. Considering that the ESL abstracts away most of the RTL design specifics, this is to be expected. ESL libraries, on the other hand, provide a wide range of APIs that cover a wide range of HW/SW coding styles, making the development process easier. Using AOP and ESL together ensures that the original code cannot be changed. The original ESL paradigm incorporates the cryptography injection/detection issues as part of the compilation and execution processes. The cryptographic ESL model, injection flaws, and detection modules are all newly split after the simulation's end. According to ESL literature, a decrease in simulation speed is also predicted. However, the uniqueness is that the underlying security attack simulation is accelerated while maintaining the same level of efficiency in terms of injection location coverage. Indeed, the distinction between the two modeling levels is one of electrical features rather than cryptographic technique. The environment for estimating a KECCAK algorithm resistance performs the identical functions at both levels, but more quickly at ESL.

6 Conclusion

Using the ESL fault injection/detection environment, we have been able to evaluate the KECCAK fault attacks security, which we describe in this paper. The suggested SystemC-AOP technique for testing the robustness of KECCAK model against fault attacks was proposed. Then, we demonstrated how to use the AOP approach to insert faults into SystemC designs at the ESL Level.

The simulation results demonstrate the suggested fault injection/detection technology is feasible. Our environment is ideal for evaluating a KECCAK design's resilience against fault injection attacks. Additionally, the simulation findings indicate that the AOP has a little effect on simulation time. The AOP technique is thus very beneficial in the test KECCAK security, since it significantly minimizes the work and error associated with security attack simulation.

Acknowledgement: The authors extend their appreciation to the Deputyship for Research & Innovation, Ministry of Education in Saudi Arabia for funding this research work through the Project Number (IF-PSAU-2021/01/18168).

Funding Statement: This research work was funded by the Deputyship for Research & Innovation, Ministry of Education in Saudi Arabia through the Project Number (IF-PSAU-2021/01/18168).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] V. Herdt and H. M. Le, D. Große and R. Drechsler, “Verifying SystemC using intermediate verification language and stateful symbolic simulation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1359–1372, 2018.

[2] T. Markwirth, R. Jancke and C. Sohrmann, “Dynamic fault injection into digital twins of safety-critical systems,” in *Proc. Design, Automation & Test in Europe Conf. & Exhibition*, Grenoble, France, pp. 446–450, 2021.

[3] J. Roux, V. Beroulle, K. Morin-Allory, R. Leveugle, L. Bossuet *et al.,* “High level fault injection method for evaluating critical system parameter ranges,” in *Proc. IEEE Int. Conf. on Electronics, Circuits and Systems*, Glasgow, UK, pp. 1–4, 2020.

[4] J. T. Xiao, T. S. Hsu, C. M. Fuchs, Y. T. Chang, J. J. Liou *et al.,* “An ISA-level accurate fault simulator for system-level fault analysis,” in *Proc. IEEE Asian Test Symp.*, Penang, Malaysia, pp. 1–6, 2020.

[5] D. Lohmann, A. Huf, D. Lettnin, F. Siqueira and J. L. Güntzel, “A Domain-specific language for automated fault injection in SystemC models,” in *Proc. IEEE int. Conf. on Electronics, Circuits and Systems*, Bordeaux, France, pp. 425–428, 2018.

[6] H. Mestiri, Y. Lahbib, M. Machhout and R. Tourki, “An AOP-based fault injection environment for cryptographic systemc designs,” *Journal of Circuits, Systems and Computers*, vol. 24, no. 1, pp. 1–2, 2015.

[7] H. Mestiri, I. Barraj and M. Machhout, “AES high-level SystemC modeling using aspect oriented programming approach,” *Engineering, Technology & Applied Science Research*, vol. 11, no. 1, pp. 6719–6723, 2021.

[8] H. Mestiri, I. Barraj and M. Machhout, “Analysis and detection of errors in keccak hardware implementation,” in *Proc. IEEE int. Conf. on Design & Test of Integrated Micro & Nano-Systems*, Sfax, Tunisia, pp. 1–6, 2021.

[9] P. Luo, K. Athanasiou and Y. Fei, “Algebraic fault analysis of SHA-3 under relaxed fault models,” *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 7, pp. 1752–1761, 2018.

[10] H. Mestiri, I. Barraj and M. Machhout, “A high-speed KECCAK architecture resistant to fault attacks,” in *Proc. IEEE Int. Conf. on Microelectronics*, Aqaba, Jordan, pp. 1–4, 2020.

[11] H. Mestiri, I. Barraj, A. A. Mohamed and M. Machhout, “An efficient AES 32-bit architecture resistant to fault attacks,” *Computers, Materials & Continua*, vol. 70, no. 2, pp. 3667–3683, 2022.

[12] J. Singh and D. P. Mohapatra, “Dynamic slicing of concurrent AspectJ programs: An explicit context-sensitive approach,” *Journal of Software : Practice and Experience*, vol. 48, no. 1, pp. 233–260, 2017.

[13] A. Przybyłek, “An empirical study on the impact of AspectJ on software evolvability,” *Empirical Software Engineering*, vol. 23, pp. 2018–2050, 2018.

[14] T. Lewarski, A. Poniszewska-Maranda, P. Veselý and M. Mikolášik, “Aspect programming with the use of AspectJ,” in *Developments in Information & Knowledge Management for Business Applications*, Manhattan, New York, USA: Springer, Cham, vol. 330. pp. 487–554, 2021.

[15] N. Chatterjee, S. Bose and P. P. Das, “Dynamic weaving of aspects in C/C++ using PIN,” in *Proc. of int. Conf. on High Performance Compilation, Computing and Communications*, Kuala Lumpur, Malaysia, pp. 55–59, 2017.

[16] M. Golasowski, J. Bispo, J. Martinovič, K. Slaninová and J. M. P. Cardoso, “Expressing and applying C++ code transformations for the HDF5 API through a DSL,” in *IFIP Int. Conf. on Computer Information Systems and Industrial Management*, Bialystok, Poland, pp. 303–304, 2017.

[17] Y. Gao, L. Chen, G. Shi and F. Zhang, “A comprehensive detection of memory corruption vulnerabilities for C/C++ programs,” in *Proc. IEEE int. Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, Melbourne, Australia, pp. 354–360, 2018.

[18] G. J. Duck and R. H. C. Yap, "EffectiveSan: Type and memory error detection using dynamically typed C/C++," in *Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Philadelphia, USA, pp. 181–195, 2018.

[19] D. Friesel, M. Buschhoff and O. Spinczyk, "Annotations in operating systems with custom AspectC++ attributes," in *Proc. of the 9th Workshop on Programming Languages and Operating Systems*, Shanghai, China, pp. 36–42, 2017.

[20] M. L. P. Souza and F. F. Silveira, "A model-based testing method for dynamic aspect-oriented software," in *Computational Science and its Applications*, Manhattan, New York, USA: Springer, Cham, vol. 10409. pp. 95–111, 2017.

[21] W. Gabsi, B. Zalila and M. Jmaiel, "Extension and adaptation of an aspect oriented programming language for real-time systems," *International Journal of Business and Systems Research*, vol. 14, no. 2, pp. 139–161, 2020.

[22] A. Muñoz, J. Toutouh and F. Jaime, "A review of dynamic verification of security and dependability properties," in *Artificial Intelligence and Security Challenges in Emerging Networks*. Hershey, Pennsylvania, USA: IGI Global, pp. 162–187, 2019.

[23] U. T. Gabor, C. C. V. Egidy and O. Spinczyk, "Interface injection with AspectC++ in embedded systems," in *Proc. IEEE 19th Int. Symp. on High Assurance Systems Engineering*, Hangzhou, China, pp. 131–138, 2019.

[24] T. Strauch, "An aspect and transaction oriented programming, design and verification language (PDVL)," in *Proc. IEEE Euromicro Conf. on Digital System Design*, Vienna, Austria, pp. 30–39, 2017.

[25] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. M. Gritschneder *et al.,* "The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems," in *Proc. IEEE Design, Automation & Test in Europe Conf. & Exhibition*, Grenoble, France, pp. 1698–1707, 2015.

[26] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre *et al.,* "LARA: An aspect-oriented programming language for embedded systems," in *Proc. of the 11th Annual Int. Conf. on Aspect-Oriented Software Development*, Potsdam, Germany, pp. 179–190, 2012.

[27] M. Leite and M. A. Wehrmeister, "System-level design based on UML/MARTE for FPGA-based embedded real-time systems," *Design Automation for Embedded Systems*, vol. 20, pp. 127–153, 2016.

[28] Y. Harada, H. Ishikawa, M. Yoo and T. Yokoyama, "A distributed multicore real-time operating system family based on aspect-oriented programming," in *Proc. IEEE int. Conf. on Industrial Technology*, Lyon, France, pp. 1389–1394, 2018.

[29] V. Obrizan and T. Soklakova, "Multiversion parallel synthesis of digital structures based on SystemC specification," in *Proc. IEEE East-West Design & Test Symp.*, Yerevan, Armenia, pp. 1–6, 2016.

[30] N. Veeranna and B. C. Schafer, "S3CBench: Synthesizable security SystemC benchmarks for high-level synthesis," *Journal of Hardware and Systems Security*, vol. 1, pp. 103–113, 2017.