On Analyzing Program Behavior Under Fault Injection Attacks

Jakub Breier
Physical Analysis and Cryptographic Engineering
Nanyang Technological University,
Singapore
jbreier@ntu.edu.sg

***Abstract*—Fault attacks pose a serious threat to cryptographic algorithm implementations. It is a non-trivial task to design a code that minimizes the risk of exploiting the incorrect output that was produced by inducing faults in the algorithm execution process.**

In this paper we propose a design of an instruction set simulator capable of analyzing the code behavior under fault attack conditions. Our simulator is easy to use and provides a valuable insights for the designers that could help to harden the code they implement.

***Keywords*-fault attacks, fault simulator, instruction set simulator, code analysis**

I. Introduction

Fault attacks on integrated circuits, first proposed in 1997 by Boneh et al. [1], are a subset of physical attacks. Usually, they exploit properties of the cryptographic algorithm that is implemented in the device, allowing the attacker to distinguish the secret key used for encryption. There are various ways of determining the key, each with different properties and suitable for different fault models and ciphers. The most popular method is Differential Fault Analysis (DFA) [2], used mostly for symmetric block ciphers. In DFA, the attacker tries to inject one or more faults in the last rounds of encryption process and then compares faulty and original ciphertexts. This process is depicted in Figure 1. Other methods include Collision Fault Analysis (CFA), Ineffective Fault Analysis (IFA), Safe-Error Analysis (SEA), and Fault Sensitivity Analysis (FSA) [3].

In order to be able to use these methods, the attacker needs to disturb the device during the computation. This can be done in various ways, by different fault injection techniques. These range from very basic, inexpensive ones, such as varying the supply voltage, to advanced techniques, requiring device de-packaging and significant funds, such as laser fault injection.

Designing the experiment and getting plausible results is usually a long-term process, depending on the device, the equipment, and the fault model. Therefore, it is beneficial to model the fault behavior of the implemented algorithm to distinguish what fault models are possible and what is the probability of a particular fault occurrence.

The same holds for the other side – when designing a protection against these attacks, the security analyst needs to test the implementation for vulnerabilities. When it comes to countermeasures, there are two possible ways to protect the implementation. Either we try to protect the device itself, by adding sensor or employing circuit-level countermeasures or by checking the software code for vulnerable points that can be exploited and fixing these points.

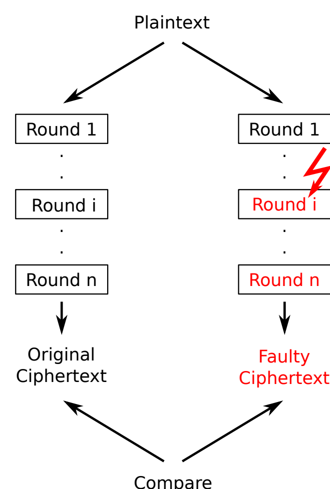

Figure 1. Schematic of the Differential Fault Analysis.

In this paper, we propose a novel instruction set simulator (ISS), based on Java programming language, able to simulate a fault behavior of software implementations for microcontrollers. Our simulator takes an assembly code as the input and checks all the possible fault models that may occur in the device. Based on this checking, it outputs an information about vulnerable instructions.

The rest of the paper is structured as follows. Section II provides an overview of the related work in this field and proposes requirements for our simulator. Section III details the design of the ISS. Fault simulation capabilities and methodology are described in Section IV. Our method is then explained on a case study, presented in Section V. Finally, Section VI concludes this paper and provides motivation for the future work.

II. Related Work

In this section, we will provide an overview of works aiming at instruction set simulators. By inspecting these works, we will state the requirements for our simulator.

In [4], the authors adjust SID1 instruction set simulator to simulate effects of time-domain electromagnetic (EM) interference in a microcontroller. They used PIC C and PIC assembly to write their code. From the component

point of view, simulator consisted of CPU with interrupt controller and external oscillator, bus component and external memory. They estimated the EM emanation caused by particular instructions and fed the simulator with these values. As a result, they could predict different program behavior with respect to EM interference.

Authors of [5] created a high performance software framework based on multi-level hash table to enable development of more efficient ISS. They classified the instructions in the instruction set to construct a hash table and used a preprocessor to map relationships between instructions and hash table elements.

In [6], the authors use their own ISS in order to simulate source code of various cryptographic algorithms implementations on 8-bit microcontroller, allowing them an easy analysis and debugging. Based on these results, they could make the implementations more efficient by utilizing various extended instruction sets.

Simulating the fault behavior of instructions was previously used in [7], where authors implemented and analyzed 19 different strategies for fault attack countermeasures. They created their custom ISS for simulating ARM Cortex M-3 and performed benchmarking allowing them to quantitatively compare the countermeasures. However, they did not provide any details about their simulator.

Based on analysis of previous works and our goals, we have stated the requirements for our fault ISS. It has to be able to:

- simulate execution of assembly code for various microcontrollers with different instruction sets and register sizes,
- inject one or multiple bit faults in the execution of chosen instruction,
- inject random byte faults and stuck-at faults in registers,
- inject single and multiple instruction skip and instruction change faults
- evaluate the faulty behavior by using all the possible inputs and all the possible faults with respect to given assembly code and indicate vulnerable instructions in the code,
- provide an easy-to-understand output that can be used for improving the code.

III. ISS Design

Before implementing fault injection capabilities, we have to design the instruction set simulator for a general-purpose microcontroller. This section provides overview on mapping of particular hardware components to object model in software. Design of our solution is depicted in Figure 2. Black components on the left side are the basic components of the Harvard architecture microcontroller [8]. On the right side, we can see a high-level class diagram of the simulator. In the following, we will explain each entity in the diagram:

- **MuC:** Microcontroller class encapsulates all the other entities constituting the device. It acts as a microcontroller itself, containing the instruction set, registers, data memory and allowing performing operations on these.
- **Instruction:** It is an abstract class, defining execute() method that is further specified by its subclasses. *MuC* contains a list of *Instruction* classes, loaded from a text file this file acts as a program memory.
- **Instruction subclasses (MOV, ADD,. . .):** Our design makes it easy to add new instructions simply by adding new subclasses of the Instruction class. This allows to simulate different architectures by using the same ISS.
- **Registers:** Registers are simulated as an array of integers. Since the majority of IoT devices contain chips with constrained hardware, register sizes are either 8 or 16 bits, therefore integer variables are enough for this purpose.
- **Memory:** Memory is simulated as a map, so that an instruction can define a variable name that serves as a key and links the value together with this key.

Text file contains the assembly code for the microcontroller and it is read and analyzed by the *MuC* class in order to assemble a program. The same class allows to run this code as well. An example of such file can be seen in Table I. The first instruction is `LDI` (load immediate) and loads the value of *'a'* into register `r0`. The second instruction does the same with a different value and a different register. The third instruction computes an xor of the values in those two registers and store the result in register `r0`. The last instruction stores the value in register `r0` in the memory, using the key *'X'* as a variable name.

Table I
Asembly text file example.

Mnemonics	Operand 1	Operand 2
LDI	r0	*a*
LDI	r1	*b*
EOR	r0	r1
ST	X	r0

IV. Fault Behavior Simulation

In order to add fault simulation capabilities to our ISS, first we have to analyze, what types of faults we want to simulate. Not all the faults are interesting for the fault analysis and also, we should only include fault models that are feasible to obtain by standard fault injection techniques.

Our fault injection methodology is depicted in Figure 3. We will further explain each component of the picture in the following:

- **Input:** In assembly programs, variables are usually loaded either from the memory or as a constants (e.g., using `LDI` instruction as in Table I). For testing the vulnerability against faults, we often have to try all the possible inputs. Therefore, the simulator allows to pre-load the inputs automatically in chosen registers without having to change the assembly code.
- **Faulty Output:** After every testing iteration, the faulty output is tested by the **Comparator**. The tester

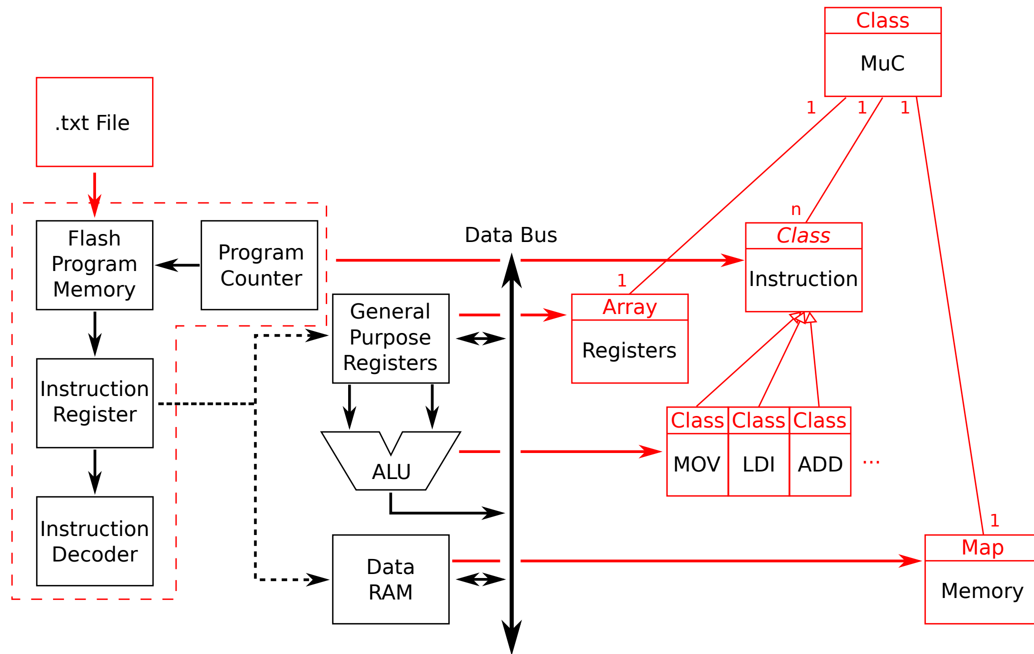

Figure 2. Microcontroller architecture mapped to an object oriented computer program. Black parts indicate physical parts of the microcontroller, red parts constitute a class diagram of the program.

can set-up this component to check for certain types of faults, depending on is he aiming for. For example, in parity check countermeasure, it can be set-up to check only the even number of bit flips in the output in order to keep the parity scheme working, however, with a faulty result.

- **Target Assembly Code:** The ultimate goal of the simulator is to test the assembly code. This code is fed to the program as a text file and can be first checked line by line if it works properly before it is tested.
- **Fault Position:** Our simulator checks all the possible position for the fault to be injected. We check every instruction and every bit in the destination register that ensures that all the bits used in the code will be tested.
- **Fault Model:** For every input and every position in the code, several fault models are tested. We have identified following fault models as the most commonly used in literature [9]:
  - *Bit flip* – this is, together with the random byte fault, the most commonly used fault model when it comes to attacking cryptographic algorithms. The simulator tests every bit in the destination register of an instruction, using single and multiple bit flips, up to the number of bits used by the register.
  - *Random byte fault* – this fault model expects flipping of random number of bits in the destination register. Because in the previous we already test all the possible combinations, we do not have to use both tests at the same time. However, random byte fault is a weaker assumption for fault attacks (it is easier to achieve in a real device and it is harder to design an attack that recovers the secret key just from a random byte flip), therefore, if an attacker only needs this fault model, he can skip the bit flip testing in order to save time.
  - *Instruction skip* – this fault model is not that popular in theoretical works, since they usually do not analyze concrete implementations of algorithms. However, as it was shown in [10], it is relatively easy to achieve this type of fault in microcontrollers and if used properly, this attack is very powerful. We test both single and multiple instruction skips, depending on the settings required by the tester.
  - *Instruction change* – for this fault model, one has to define the opcodes of used instructions. It is then possible to set a bit flip model on the opcode that changes one instruction to a different one. Again, this can be either single or multiple bit flip. For example, let the opcode of `AND` instruction be `001000` and the opcode of `EOR` instruction be `001001` (these are the opcodes used by Atmel 8-bit AVR devices). One can easily see, that if the last bit of the opcode is flipped (in other words, the fault mask is `000001`), one instruction will change to another and vice versa. Again, this can provide the attacker a powerful, yet implementation specific, fault model.
  - *Stuck-at fault* – this fault model changes the value in register to some specific value. Authors of [11] have shown that with different laser energy, it is possible to force certain memory bits either to *'0'*, or to *'1'*, allowing precise stuck-at

faults. Again, we test all the destination registers in the target code for stuck-at faults specified by the tester.

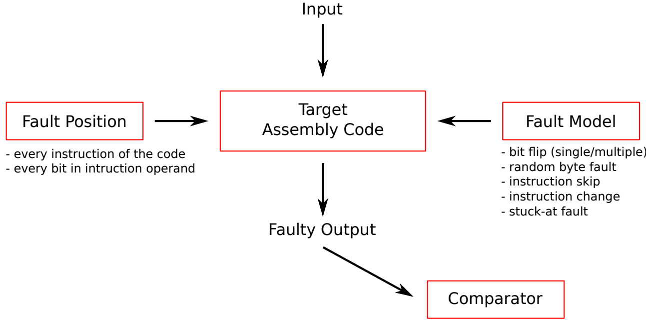

Figure 3. Fault injection methodology for ISS.

The **Comparator** provides a human-readable output as a result of the code analysis. This output is in two forms overview and a detailed view.

Overview shows the total number of faults and this number is then further divided by the tester's requirements. Usually, it is desired to have the output in some special form, e.g. specific encoding or some fault mask. The tester can then specify the output to be divided in two subsets one that fulfils the requirement and the other that does not.

Detailed view provides insight on all the successful faults, i.e. only on those in the subset that fills the tester's requirements. Table II shows all the fields that are provided by the detailed view for each fault model. Please note that the output from faulty execution is always provided in any fault model that is selected.

Table II
FIELDS FOR DIFFERENT FAULT MODELS PROVIDED IN THE DETAILED VIEW.

Fault Model	Fields
Bit flip	Instruction number, Instruction mnemonics, Fault mask, Number of plaintexts affected
Random byte fault	Instruction number, Instruction mnemonics, Fault mask, Number of plaintexts affected
Instruction skip	Instruction number, Instruction mnemonics, Number of plaintexts affected
Instruction change	Instruction number, Instruction mnemonics, Instruction opcode, Fault mask, New instruction mnemonics, New instruction opcode, Number of plaintexts affected
Stuck-at fault	Instruction number, Instruction mnemonics, Stuck-at mask, Number of plaintexts affected

Activity diagram for the whole process of code analysis is stated in Figure 4. After writing the code it is necessary to check if the instruction set used is also implemented in the fault simulator. If not, the tester has to implement missing instructions. Because of the modular architecture of our framework, it is easy to add new instructions. Afterwards, he defines which fault models he want to test the code against and prepares a set of inputs, since in some cases it is not necessary to test all the possible inputs. Before running the simulator, he has to specify the output format and in some cases also a set of outputs that constitute a security risk so that the **Comparator** could classify the outputs correctly. After getting the results, the tester should analyze the vulnerable instructions and propose changes in the code before re-running the simulations again.

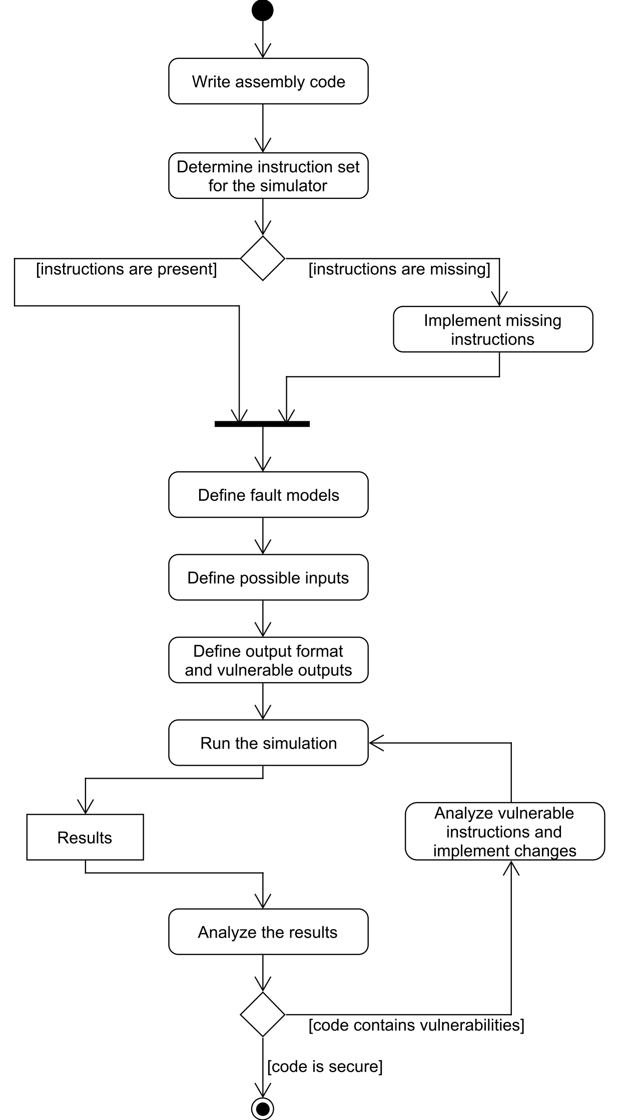

Figure 4. Activity diagram for particular steps in the code design and analysis.

V. Case Study

In this section, we will present a simple test case that shows how our simulator works.

Let us assume we are using an 8-bit microcontroller and in order to prevent the side-channel leakage, we decided to use only the codewords with Hamming weight 2. For the sake of simplicity, let us assume we only use the last four bits of the word, which gives us six possible values (`00000011`, `00000101`, `00000110`, `00001001`, `00001010`, `00001100`). Therefore, if

the attacker wants to perform a successful attack, he has to change the original output in a way that still preserves the encoding, otherwise the output checker would discard the value.

The code for performing a binary operation • on codewords is stated in Table III. It uses a look-up table to get the result of the operation from the memory. First, it loads values *'a'* and *'b'*, then it shifts *'b'* four bits left and joins these two values together with `OR`. Finally, `LDD` is used to load the value at the memory address specified by `r0` and to put the value in `r2`. Look-up table returns all-zero output in case of an invalid address. The full table for • operation is listed in Table IV. One can easily see that it is actually a cyclic group $\mathbb{Z}_6^+$.

Table III
ASSEMBLY CODE FOR OUR EXAMPLE.

#	Instruction
0	LDI r0 *a*
1	LDI r1 *b*
3	LSL r1 4
4	OR r0 r1
5	LDD r2 r0

Table IV
RESULTS FOR BINARY OPERATION •. FIRST FOUR ZERO BITS ARE OMITTED TO SAVE THE SPACE.

•	0011	0101	0110	1001	1010	1100
0011	0011	0101	0110	1001	1010	1100
0101	0101	0110	1001	1010	1100	0011
0110	0110	1001	1010	1100	0011	0101
1001	1001	1010	1100	0011	0101	0110
1010	1010	1100	0011	0101	0110	1001
1100	1100	0011	0101	0110	1001	1010

We analyzed the code in the fault simulator against following fault models: single and double instruction skip, single and double bit flip, stuck-at `00000000` fault, and stuck-at `11111111` fault. Because of the properties of the look-up table, most of the faults will force the output to zero, excluding a few special cases. One can easily see this implementation is vulnerable against even bit flip attacks. We have tested double bit flips that are a subset of these attacks, and the results show that every instruction of the code is vulnerable to this fault model. From the total of 5040 faults (number of possible plaintexts $\times$ number of instructions $\times$ bit combinations), 17.14% resulted into correct encoding with a different value than the original result.

After the result, the analyst has to decide how to deal with the vulnerabilities. For example, one of the common techniques is a modular redundancy – using more space to store the values and perform some operations multiple times. The most naïve approach would be to use two more registers where the original data would be copied, double instructions 3-5 and use `AND` to join the values in the end. This countermeasure would lower the chance of a successful attack to 8.57%. In case we even load all the data twice, the chance lowers to 2.86%. Other techniques would include using more sophisticated algebraic operations or different look-up tables.

VI. Conclusion

In this paper we presented an instruction set simulator capable of analyzing code vulnerabilities with respect to fault attacks. We designed a modular framework that allows adding new intruction sets easily and also allows specifying fault models to be tested. Assembly code is fed to the simulator in a text file in order to provide a detailed analysis of each instruction. This output can help the designer to improve the code and minimize the risk of successful fault attack.

In its current state, the simulator supports execution of the most used instructions of assembly language for 8-bit Atmel AVR microcontrollers. We were able to successfully simulate software encoding schemes that provide side-channel resistance for cryptographic implementations and harden these schemes against fault injection attacks [12].

For the future work, we would like to enhance our simulator to provide a deeper analysis that would suggest improvements based on used instructions and valid format of inputs and outputs. The simulator would be able to decide which instructions can be replaced with safer variants and whether some parts of a code can be interchanged with look-up tables. Also, the simulator would estimate the overhead of such changes.

References

[1] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'97. Berlin, Heidelberg: Springer-Verlag, 1997, pp. 37–51.

[2] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology - CRYPTO '97*, ser. Lecture Notes in Computer Science, J. Kaliski, BurtonS., Ed. Springer Berlin Heidelberg, 1997, vol. 1294, pp. 513–525.

[3] J. Breier and D. Jap, "A survey of the state-of-the-art fault attacks," in *Integrated Circuits (ISIC), 2014 14th International Symposium on*, Dec 2014, pp. 152–155.

[4] S. Y. Yuan, H. E. Chung, and S. S. Liao, "A microcontroller instruction set simulator for emi prediction," *IEEE Transactions on Electromagnetic Compatibility*, vol. 51, no. 3, pp. 692–699, Aug 2009.

[5] Z. Hao, P. Chu, T. Zhang, D. Wang, and C. Hou, *Recent Advances in Computer Science and Information Engineering: Volume 5*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. A High-Performance Framework for Instruction-Set Simulator, pp. 9–14.

[6] H. Groß and T. Plos, *Radio Frequency Identification. Security and Privacy Issues: 8th International Workshop, RFIDSec 2012, Nijmegen, The Netherlands, July 2-3, 2012, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. On Using Instruction-Set Extensions for Minimizing the Hardware-Implementation Costs of Symmetric-Key Algorithms on a Low-Resource Microcontroller, pp. 149–164.

[7] N. Theissing, D. Merli, M. Smola, F. Stumpf, and G. Sigl, "Comprehensive analysis of software countermeasures against fault attacks," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 404–409.

[8] I. Yasui and Y. Shimazu, "Microprocessor with harvard architecture," Jul. 23 1991, uS Patent 5,034,887. [Online]. Available: https://www.google.com/patents/US5034887

[9] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, Nov 2012.

[10] J. Breier, D. Jap, and C.-N. Chen, "Laser Profiling for the Back-Side Fault Attacks: With a Practical Laser Skip Instruction Attack on AES," in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security*, ser. CPSS '15. New York, NY, USA: ACM, 2015, pp. 99–103.

[11] F. Courbon, P. Loubet-Moundi, J. Fournier, and A. Tria, "Adjusting laser injections for fully controlled faults," in *Constructive Side-Channel Analysis and Secure Design*, ser. Lecture Notes in Computer Science, E. Prouff, Ed. Springer International Publishing, 2014, pp. 229–242.

[12] J. Breier, D. Jap, and S. Bhasin, "The other side of the coin: Analyzing software encoding schemes against fault injection attacks (to appear)," in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, May 2016, pp. 1–8.