Hardening of Smart Electronic Lock Software against Random and Deliberate Faults

Jakub Lojda, Richard Panek, Jakub Podivinsky, Ondrej Cekan, Martin Krcma, Zdenek Kotasek
Faculty of Information Technology, Brno University of Technology, Centre of Excellence IT4Innovations
Bozetechova 2, 612 66 Brno, Czech Republic
Email: {ilojda, ipanek, ipodivinsky, icekan, ikrcma, kotasek}@fit.vutbr.cz

***Abstract*—In this research paper, analysis of smart electronic lock behavior during presence of faults in its controller is examined. A typical smart electronic lock is composed of a controller unit, usually implemented in a processor, and the mechanical part, which may be for example a stepper motor. The goal of this research paper is to examine the consequences of failing controller running a partly hardened program, which we developed from the experiences we gained in our previous research. We implement the controller processor in *Field Programmable Gate Array* (FPGA) in order to inject faults into our components. This paper focuses on fault injection into occupied parts of *Instruction Memory* (IMEM) and *Data Memory* (DMEM). Moreover, permanent failures of the processor are simulated by fault injection into occupied *Look-up Tables* (LUTs) of the processor design on the FPGA. Our results show that the application of certain SW-implemented fault tolerance methods may, in opposite, degrade the hardness of the system. Our experiments imply that the IMEM is the most sensitive to fault injection, because there is no possibility for an eventual self repair. In the case of DMEM, erroneous values may be possibly repaired when the variable is rewritten back to the memory, slightly lowering the DMEM sensitivity to fault injections. The CPU itself is the least susceptible. Although faults are injected to the utilized contents only, for the CPU LUTs, a certain part of the logic may not be used to implement the required function.**

***Keywords*—Electronic Lock, Stepper Motor, Fault Tolerance, Fault Injection, FPGA, IMEM, DMEM, LUT.**

I. Introduction

An electronic lock (also called smart lock) [1] belongs to the family of the so-called smart devices. It is a cyber-physical system that is used to secure access in a very similar way as an ordinary door lock. The difference is in the authorization of the user, for which the electronic lock uses other modern approaches that are based for example on biometrics [2], the smart lock can also be connected to a local network [3]. A failure of electronic lock can cause high losses, whether it is unauthorized unlock or unauthorized lock, denial of authorized unlock etc. Therefore it is important to research reliability of such electronic locks whether it is their failure or intentional tampering in order to attack the electronic lock. Generally, reliability of electronic systems can be achieved by two main approaches: 1) *Fault Avoidance* [4], in which faults are avoided through the selection of reliable components; and 2) *Fault Tolerance* (FT) [5], in which faults are accepted as a fact and the higher reliability is achieved through structural modifications of the original system.

Various studies of smart lock reliability and security can be found in the literature. For example authors of paper [6] present a security analysis of a particular smart lock. According to the author's findings, the selected lock was prone to several security vulnerabilities in the back-end services of the smart lock. In the paper [7], security problems of the *Internet of Things* (IoT) are discussed and presented on the example of a smart lock. The research presented in [8] deals with security analysis of another particular smart lock. Authors of [9] present an extensive security evaluation of a smart home application writing framework, thus focusing on the application and server part of the system. The authors managed (among others) to obtain existing lock codes and disable vacation mode.

From the previous paragraph it is obvious, that today's consumer electronic locks may, and usually do, contain many security and safety problems. However, in this research paper, we focus on the electronic part of the smart lock mainly. Various reactions to faults injected into memories (i.e. IMEM with program instructions and DMEM with data) and also processor logic are observed and evaluated. For the realization, SRAM-based FPGA chips are used, as these enable us to rewrite the configuration SRAM memory in order to modify the design during run time and intentionally introduce faults into the HW.

This paper is organized as follows. The evaluation platform we utilized for monitoring faults impacts in electro-mechanical systems is presented in Section II. Experimental evaluation of faults injected into SW controller program (stored in IMEM) and run-time data (stored in DMEM), alongside with injection into HW logic in LUTs, is presented in Section III. Section IV concludes the paper and presents ideas for our future research.

II. The Evaluation Platform

In our previous paper [10], we presented a platform for faults effect on electro-mechanical system evaluation. The goal of our platform is to monitor the behavior of both the electric and the mechanical part of the system and evaluate whether an injected fault had an effect on any of the parts or whether it even caused a complete failure of the whole system. To achieve this goal it was convenient to use functional verification [11] as the base for the platform.

In our previous work we have developed a fault injector which allows us to inject faults into a specified position in a bitstream stored in the configuration memory. Using the RapidSmith [12] we are able to determine which bits in the bitstream are related to the particular parts of the implemented design. We focus primarily on injection to *Block Random Access Memories* (BRAMs) and LUTs.

A. Stepper Motor Control Using a Processor on the FPGA

We decided to use the NEO430 [13] soft-core processor to implement the motor controller in the FPGA. The NEO430 processor uses separate memories to store a program (IMEM) and data (DMEM). It is equipped with multiple peripheral interfaces. We used the *Custom Functional Unit* (CFU) interface to enter the number of the motor steps and the *General-purpose Inputs and Outputs* (GPIOs) to read the output data. The final version of the program uses the application image

978-1-7281-9535-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSD51259.2020.00110

saved in boot *Read-only Memory* (ROM), therefore there is no need to load up the program from an external source. The program controls the stepper motor using the signals SIG_A, SIG_B, SIG_C and SIG_D, the number of steps is defined by the input signal STEPS. The whole controller schematics is shown in Figure 1. The three red frames in the figure mark units targeted by the fault injection: 1) IMEM; 2) DMEM; and 3) LUTs implementing the processor logic.

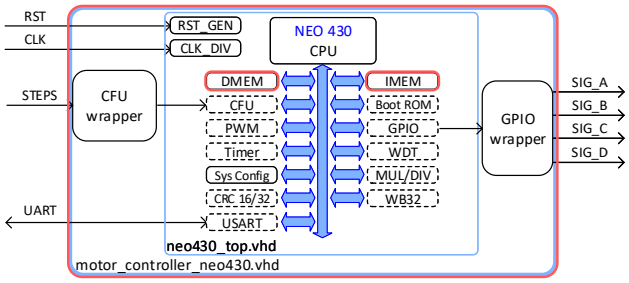

Figure 1: The use of soft-core processor NEO430 as the main part of stepper motor controller.

B. Electronics Controller Output Software Data Processing

The software part of our platform is executed independently on a personal computer. The most important part of the software is the stepper motor simulation which evaluates the faults effects on the mechanics. We monitor the angle of the motor rotation i.e. whether the required angle was achieved or whether an anomaly occurred in the motor behavior. In our experiments, we used the MATLAB and the Simulink [14] software, specifically the Simscape [15] library to perform the simulations. This library offers a customizable simulator of a stepper motor. We utilized the adjustable parameters to define a 4-phase stepper motor with a permanent rotor. The motor is equipped with 1/64 transmission gearbox and single step angle of 5.625/64. It is needed to perform 4,096 steps (64 steps without the gearbox) for the full rotation. The simulation output describes the current rotation angle and the final angle.

III. Experiments and Experimental Results

Three areas of the fault injection were identified: **1) IMEM** – storage of the instruction memory; and **2) DMEM** – storage of the run-time data, because memories are most susceptible to a failure; moreover **3) LUTs** occupied by the processor HW on the FPGA which are used to approximate appearance of permanent faults in the processor logic itself. For the IMEM and processor LUT, two fault injection strategies were selected: **I) single fault injection**, during which only one single bit flip fault is injected into the given configuration memory contents before the design is activated; and **II) multiple fault injection**, during which random faults are injected periodically into the configuration memory contents. For the DMEM memory, single fault injection is not justified, because the DMEM contents is constructed during the run time, thus rewriting any discrepancies made to the memory contents before the design was started. For this reason, the DMEM was tested to multiple faults injection only. The faults were injected into utilized bits exclusively.

A. Controller Processor Programs

Three program versions for the processor were created and tested using fault injection. The first program was *"original"*, the unhardened one. Based on our previous experiments from our previous paper [16], two possibly weak points in the program were identified. We hardened these parts of the program by inserting certain redundant structures. These modified programs include: **A)** After output pin write, the algorithm waited for next output configuration. In this modification, these values on the output pins of the processor are constantly written during the *delay* function execution. This should prevent a long-term discrepancy on the output pins of the processor, if the output part was hit by fault. **B)** The output pin signals, which are prepared in the memory, may be altered by a memory failure. For this reason, in the *"B"* version of the program, these values are stored in three copies in the memory and bit-voted before each output step preparation. This version also includes modifications from the version *"A"* (i.e. the values are constantly written during *delay* function execution).

B. Searching for Occupied Bytes of Data Memory

Waiting for failures to appear would take very long time if we injected faults into the complete address space of the DMEM, which is 2 KiB in size. This is why we injected faults into the used DMEM cells only. Also, from our results, average occupation of DMEM space was 39 B, which is 1.9 %. The used bits must be obtained during the run time of the system. While some data in the DMEM stay at equivalent position during the run or even between restarts of the system, other data may be dynamically allocated during the run time and there is no guarantee the data will remain at the same address. This is the reason why we use this empirical method, which is, nevertheless, supported by a considerable amount of analyzed data. As the speed of DMEM readback is limited, it is not possible to gain all the states of DMEM from one run. This is the reason why we repeat this procedure with slight deviations between periodic readbacks in order to cover the most states of DMEM. Approximately 1,000 readbacks per program were made. Each 1 B of DMEM was marked as used if a non-zero value was detected at any time during its readback.

C. Parameters of Tested Units

The designs were synthesized for the Xilinx Virtex 5 FPGA on the ML506 board [17]. The Xilinx *Integrated Synthesis Environment* (ISE) 14.7 [18] was used to synthesize the bitstreams. The following Table I shows the FPGA resources used by the processor, its program and data. It also illustrates the number of bits of the bitstream, into which the faults were injected intentionally.

TABLE I: The FPGA resources used by the processor (LUT), its program (IMEM) and run time data (DMEM).

Controller Unit Variant	*IMEM Occupied [b]*	*DMEM Occupied [b]*	*LUTs Used [b]*
Original	14,336	232	58,496
Variant "A"	14,848	232	58,496
Variant "B"	22,784	464	58,496

D. Single Fault Injection Testing Strategy

In these experiments, single bit flip of one bit that was selected uniformly-at-random from all the occupied bits was performed before the processor was started. The processor is configured to rotate the stepper motor for 80 s, in order to extend the amount of data obtained for the analysis of its

behavior. This scenario has to imitate a fault occurrence before the electronic lock has to be locked or unlocked. One test run for the single fault injection strategy looks as follows: **1)** the processor repaired while a new bitstream is downloaded to the FPGA; **2)** a fault into one uniformly-at-random selected bit is injected; **3)** the processor is started and instructed to perform 12.4 rotations; **4)** logical values on the output pins of the processor are observed and saved for later analysis; **5)** the run of one experiment finishes when the processor output pins stop to change or when a timeout of 220 s occurs.

First, the failures of the electronic controller were classified. These results can be observed in Table II. The first column of the table contains the percentage of failed runs from the total amount of experiments hold with the specific configuration. The next three columns classify electronic failures into one of the three specific types (i.e. *Stuck*, *Timeout* and *Mismatch*), which are also represented as a percentage. The right part of Table II shows the cases, in which the mechanical part ended in the correct position although the electronic failure was detected in the results. The first column of this part of the table (i.e. *Mechanic OK – Total*) represents the total percentage share of such cases based on the number of runs in which electronic failure was detected. Next three columns represent the percentage share of cases, in which the electronic failed and also the mechanical part ended with *Stuck*, *Timeout* or *Mismatch*. As can be observed, the number of runs with electronic failure was higher for the IMEM. The higher sensitivity to failure of the IMEM can be attributed to the injection into utilized bits only. In the CPU variant, faults are also injected into utilized LUT bits only. Not each of these bits, however, implements the required function. Thus, fault injection into some LUT bits may end up without any failure manifesting.

TABLE II: The results of single injection experiments with the failures classification.

	Electronic Failure				*Mechanic OK (Out of Electronic Failed Runs)*			
	Total [%]	*Stuck* [%]	*Timeout* [%]	*Mismatch* [%]	*Total* [%]	*Stuck* [%]	*Timeout* [%]	*Mismatch* [%]
CPU "O"	5.3	4.5	0.4	0.4	7.3	3.2	0.0	4.4
CPU "A"	5.9	5.1	0.5	0.3	8.4	3.9	0.7	3.9
CPU "B"	6.2	5.5	0.5	0.2	10.4	5.1	0.3	2.4
IMEM "O"	36.7	15.6	14.0	7.0	20.3	1.4	1.0	18.0
IMEM "A"	35.2	15.5	14.4	5.4	21.2	4.8	3.7	12.6
IMEM "B"	34.0	17.3	9.6	7.1	25.0	3.8	1.6	19.6

We can take these results into account with the particular variants of the program – unhardened (i.e. *Original*) *"O"*, hardened *"A"* and *"B"* with the extended hardening. The results imply that the actual level of hardening does not have any significant impact since the differences are in the magnitude of a few percent or even fractions of a percent. In the case of IMEM injections we can observe a slightly better results gained by the hardened programs. However, the results of the injection into the processor show an opposite tendency. According to the small differences we can infer they are a mere statistical error and the hardening did not have any effect.

Figure 2 shows a statistical box plot that illustrates a maximum number of rotations achieved during runs with the failing electronics controller. As you can see, the processor (CPU) injection usually caused the motor to stuck at zero, i.e. it did not rotate at all. This does not stand for the IMEM injection, after which the processor is able to run and it was able to achieve usually zero to 12.4 rotations, therefore to reach the desired number of rotations. This kind of faults may prevent the door to lock or unlock.

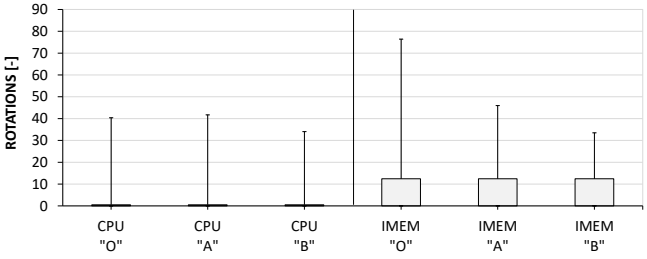

Figure 2: Box plot chart with rotation angle for single injection.

E. Multiple Fault Injection Testing Strategy

The multiple fault injection strategy examines the influences of several consequently following faults on the resulting behavior of the system. The first fault is injected into the system after the system was running for 10 s. From this time, every 5 s, one fault is injected into the system. The processor is programmed to rotate the motor for 80 s. The multiple fault injection has this scenario: **1)** the processor repaired while a new bitstream is downloaded to the FPGA; **2)** the processor is started and instructed to perform 12.4 rotations, logical values on the output pins of the processor are concurrently observed and saved for later analysis; **3)** the fault injector waits for 10 s; **4)** system clock of the processor is stopped as well as the simulation time; **5)** one single fault is injected into the system; **6)** system clock of the processor and the simulation time are started again; **7)** the fault injector waits for 5 s; **8)** if no failure was observed on the output pins and the timeout was not reached, go to (4); **9)** the run of one experiment finishes when the processor output pins stop to change or when a timeout of 220 s occurs.

The results for the multiple injection experiments are shown in Table III. For these experiments, failed runs significantly increased. This is because a fault is injected every 5 s until a failure is detected or a timeout of 220 s is reached.

TABLE III: The results of multiple injection experiments with the failures classification.

	Electronic Failure				*Mechanic OK (Out of Electronic Failed Runs)*			
	Total [%]	*Stuck* [%]	*Time-out* [%]	*Mis-match* [%]	*Total* [%]	*Stuck* [%]	*Time-out* [%]	*Mis-match* [%]
CPU "O"	71.3	14.4	24.9	32.1	16.0	2.0	1.5	12.5
CPU "A"	70.1	13.6	30.1	27.0	16.3	2.2	3.2	10.9
CPU "B"	89.0	19.6	23.4	46.0	12.1	1.9	1.0	9.2
IMEM "O"	99.1	41.1	27.1	30.9	19.7	1.8	0.2	17.7
IMEM "A"	98.4	31.8	31.4	35.3	31.7	4.4	0.2	27.1
IMEM "B"	99.7	40.1	22.5	37.1	27.3	3.9	0.2	23.3
DMEM "O"	91.8	9.0	15.4	67.4	13.1	0.0	0.0	13.1
DMEM "A"	92.8	11.6	15.6	65.6	13.8	0.0	0.0	13.8
DMEM "B"	95.4	34.2	3.0	58.2	17.2	0.0	0.0	17.2

The DMEM injection led to a higher number of failures than the CPU injection but still to a lower number than the IMEM injection. The higher IMEM vulnerability to the faults is due to its nature when the memory is only read from but never written to. Therefore, there is no possibility to a fault being repaired by rewriting the data. It is interesting however, that the *"B"* hardening level, the highest one, led to the higher number of failures. Therefore, although designed to do the

opposite, it actually made the system more vulnerable. We believe that the reason is the added logic in the SW, which makes the program more sensitive to fault injection.

The faults effects on the mechanics are shown in Figure 3, which shows that the motor achieved the rotation angle close to the required value or a bit lower, while sometimes it suffered significant deviations. We can again conclude that this way of injecting faults may prevent the door to lock or unlock.

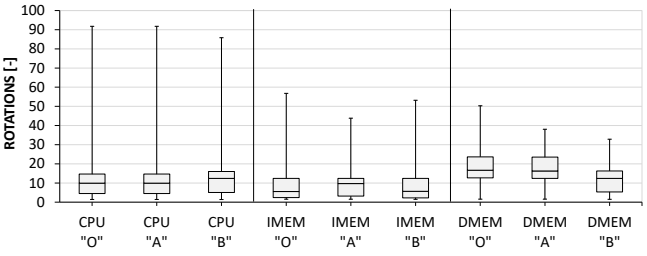

Figure 3: Box plot with rotation angle for multiple injection.

Not only the fact that a failure occurred is important, but also the number of faults that had to be injected in order to manifest the failure provides valuable information. This information is shown in the box plot chart in Figure 4. A relatively high number of fault injections into the DMEM is needed to manifest a failure, which confirms the hypothesis that the DMEM content may be eventually rewritten and thus corrected. In order to manifest a failure during the CPU experiment, a smaller number of faults is needed. On the other side, during the injection into the IMEM, the smallest number of fault injections is needed to manifest a failure, which can also be attributed to the static character of this memory type.

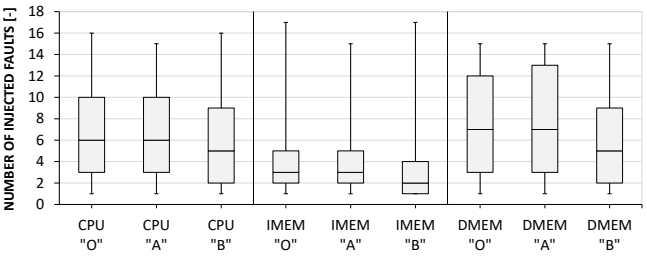

Figure 4: Box plot with number of injected faults until failure.

IV. Conclusions and Future Research

In this paper, we evaluated the reliability of an electronic lock controlled by a processor. We monitored effects of faults injected into the program memory (IMEM), data memory (DMEM) and to the processor itself as well. The results imply that if faults were injected into the controller with malicious intent to open the lock without authorization, the probability of success is very low and this attack vector is not very promising. However, if the goal was to prevent the door to lock or unlock in the first place, the chances are significantly higher and it may lead to a successful attack.

A significant part of this work was an implementation of software redundancy using *"A"*: continuous propagation of the output data to the output pin and *"B"*: triplicating variables in the data memory and following comparison using a software voter in combination with the measure *"A"*. However, our experiments did not prove any significant improvement of the system reliability when hardened using these measures. Our results show that the application of certain SW-implemented fault tolerance methods may, in opposite, degrade the hardness of the system. Therefore, the future work will be to improve these methods and find an optimal way of hardening the lock.

Acknowledgements

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science – LQ1602, the Brno University of Technology under number FIT-S-20-6309 and the JU ECSEL Project SECREDAS (Product Security for Cross Domain Reliable Dependable Automated Systems), Grant agreement No. 783119.

References

[1] Y. T. Park, P. Sthapit, and J.-Y. Pyun, "Smart digital door lock for the home automation," in *TENCON 2009-2009 IEEE Region 10 Conference*. IEEE, 2009, pp. 1–6.

[2] J. Baidya, T. Saha, R. Moyashir, and R. Palit, "Design and implementation of a fingerprint based lock system for shared access," in *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, 2017, pp. 1–6.

[3] A. Kassem, S. E. Murr, G. Jamous, E. Saad, and M. Geagea, "A smart lock system using wi-fi security," in *2016 3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, 2016, pp. 222–225.

[4] J.-C. Geffroy and G. Motet, *Design of Dependable Computing Systems*. Kluwer Academic Publishers, 2002.

[5] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[6] E. Knight, S. Lord, and B. Arief, "Lock picking in the era of internet of things," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2019, pp. 835–842.

[7] M. Pavelić, Z. Lončarić, M. Vuković, and M. Kušek, "Internet of things cyber security: Smart door lock system," in *2018 International Conference on Smart Systems and Technologies (SST)*, 2018, pp. 227–232.

[8] M. Ye, N. Jiang, H. Yang, and Q. Yan, "Security analysis of internet-of-things: A case study of august smart lock," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017, pp. 499–504.

[9] E. Fernandes, J. Jung, and A. Prakash, "Security analysis of emerging smart home applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 636–654.

[10] J. Podivinsky, O. Cekan, J. Lojda, M. Zachariasova, M. Krcma, and Z. Kotasek, "Functional Verification based Platform for Evaluating Fault Tolerance Properties," *Microprocessors and Microsystems*, vol. 52, pp. 145 – 159, 2017.

[11] A. Meyer, *Principles of Functional Verification*. Elsevier Science, 2003. [Online]. Available: http://books.google.cz/books?id=qaIiX3hYWL4C

[12] C. Lavin, M. Padilla, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapid Prototyping Tools for FPGA Designs: RapidSmith," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 353–356.

[13] S. Nolting, "NEO430 Processor," https://github.com/stnolting/neo430, 2018.

[14] MathWork®, "MATLAB and Simulink," https://www.mathworks.com/, 2018, accessed: 2019-03-20.

[15] MathWork®, "Stepper motor," https://www.mathworks.com/help/physmod/sps/powersys/ref/steppermotor.html, 2019, accessed: 2019-03-20.

[16] J. Podivínský, J. Lojda, R. Pánek, O. Čekan, M. Krčma, and Z. Kotásek, "Evaluation platform for testing fault tolerance: Testing reliability of smart electronic locks," in *2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE Circuits and Systems Society, 2020, pp. 1–4.

[17] Xilinx Inc., "Ml506 Evaluation Platform User Guide," *UG347 (v3. 1.2)*, 2011.

[18] Xilinx Inc., "ISE Design Suite," https://www.xilinx.com/products/design-tools/ise-design-suite.html, 2017, accessed: 2017-07-07.