Analysis of Software-Implemented Fault Tolerance: Case Study on Smart Lock

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 our research, we focus on Fault-Tolerant system design and testing. Recently, we also studied Fault Tolerance against random and deliberate faults of electronic smart locks. In our last research, we tested Software-Implemented Fault Tolerance in the controller of a smart electronic lock. We found out that the most sensitive part is the Instruction Memory, but also that our hardening proved to have only negligible effects on the resulting fault tolerance. In this paper, we extend our experiments and provide further analysis of potential pitfalls when hardening using SIFT. We found out that added hardness may improve resilience to faults. But also, the resilience may be instantly worsened by other factors, such as increased bus traffic. In our research we found out, that our hardening did not improve the resiliency to faults most likely due to the increased bus traffic. This means that it is always important to consider the complete system and also the parts of the system that are easily overlooked.**

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

I. Introduction

Recently, the so-called Smart Devices [1] gained their popularity. These include the so-called Smart Electronic Lock [2]. It acts as an ordinary door lock, except it can be unlocked by unordinary means, such as by a gesture on a smartphone [3]. It is obvious that a smart lock is a critical device which must satisfy certain reliability standards.

Reliability in electronic devices can be achieved in two different ways: 1) *Fault Avoidance* (FA) [4], which selects from reliable components to build the system. 2) *Fault Tolerance* (FT) [5], on the contrary, changes the structure of the system, so a component failure is not observable on the system behavior. From the FT, the so-called *Software-Implemented Fault Tolerance* (SIFT) [6] is derived, which changes SW code structure to increase its reliability.

Our research focuses on FT design and evaluation. It is important to intensively test FT systems to ensure their quality. For this purpose, the so-called Fault Injection can be used, which intentionally introduces faults into the system. During this, the system is observed and its behavior is evaluated. We hardened and evaluated an electronic lock controller in our previous paper [7]. For the purpose of evaluation, its processor was implemented in *Field Programmable Gate Array* (FPGA), which offered us the possibility to inject faults at run time. Fault injections into the *Instruction Memory* (IMEM), *Data Memory* (DMEM) and the CPU logic itself were evaluated. The results indicated, that the most sensitive is the IMEM. Our tests were held on three different programs, out of which two contained SIFT. The data showed, however, that our SIFT methods did not prove to be beneficial. In opposite, our SIFT made the systems more vulnerable. And this paper focuses on the analysis and explanation of such behavior, as we believe that identifying and avoiding such anomaly is useful in the following research. In this paper, we add a new set of experiments and analyze three additional aspects that are related to the mentioned anomaly. These include: 1) compiler program code optimization; 2) accuracy of DMEM occupied bytes detection; and 3) increase of CPU internal bus transfer rate, which could possibly explain the anomaly.

Security and safety of smart electronic locks are studied in the literature. For example authors of [8] present survey on various identification systems that are usually used in smart locks. Another paper [9] presents a detailed analysis of the security of a specific commercially available smart lock. New SIFT methods can also be found in the literature. Authors of [10] introduce and evaluate a method utilizing unused resources to implement SIFT on the Itanium 2 CPU, which is the *Explicitly Parallel Instruction Computing* (EPIC) processor. Another paper [11] presents an analytic method to evaluate reliability of multi-computer SIFT systems.

This paper is organized as follows. Electronic locks structure with discussion about their reliability is presented in Section II. Evaluation platform for monitoring faults impacts in electro-mechanical systems is presented in Section III. 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 IV. Section V presents the analysis of our results and concludes the paper.

II. Electronic Locks

Smart electronic lock is a relatively complex device that uses the latest technologies of nowadays. It typically consists of three parts (modules) [12]: 1) Control Module; 2) Motor Module; and 3) I/O Module. The management of the entire lock is provided by the Control Module which performs a number of computational extensive operations, therefore, it is typically realized by a processor. The mechanical part of the lock consists of the Motor Module, which can be realized by

978-1-7281-9899-6/20/$31.00

various drives that manipulate the lock. In our research, we focus on a stepper motor, which is very often used in smart locks as the motor [13]. The stepper motor has its rotation divided into several equal steps, allowing precise control of the position of the rotation by means of input pulses. The I/O Module is used for a communication and performs mainly the communication with interfaces such as Wi-Fi or Bluetooth.

In our research, we focus on the change of the processor data. This can be a program change – another instruction sequence is executed, or a data change – other values are used. The injection of faults into the processor may result in unexpected and unwanted behavior of the smart lock and consequently property damage. The fault can be induced naturally from environment via charged particle or through attacker which intends to change the data in memory by electromagnetic interference or by specific material that secretes these particles. The fault can also occur when attacker mechanically damages the smart lock, its circuit board or another component.

When data are corrupted, the lock can be unlocked if incorrect authentication is performed or can stay in the lock state when unlocking with the correct credentials. It may also happen that the lock is not really locked when the lock is requested. Anyone will have access to a permanently unlocked door. However, there may also be a failure that occurs only in a certain situation, i.e. only in a certain state of the lock. Such a fault is very difficult to be detected and it is not entirely clear when and what behavior will occur in the fault. Therefore, in this research, we focus on the impact of these faults in the smart lock on the processor.

III. Evaluation Platform

In our previous work we introduced a platform for fault tolerance evaluation [14]. This platform is based on functional verification principles combined with faults injection into an FPGA. Functional verification is based on the simulation of a verified system and monitoring its outputs and comparing it to a reference data after feeding predefined inputs to the system. We used this principle for our purposes, however we implement the verified system into an FPGA which allows us to easily inject faults to the system and evaluate their effects.

The platform capabilities were demonstrated on an example of a robot searching for its way through a maze. It was an FPGA controlled simulated system aimed to experimentally evaluate the platform, however the platform was designed to be scalable and able to evaluate any system controlled by an FPGA. It offers a convenient way to evaluate faults effect on the controller and the stepper motor of an electronic lock. The experimental results of this work are based on our platform.

To successfully use the platform, we have been forced to modify application specific components of the platform. It is necessary for the controller to be implemented in an FPGA. It is vital to establish a proper communication line between the control unit operating in the FPGA and the software simulating the stepper motor running on the different computing platform. For these purposes we use MATLAB and the Simulink [15] software, specifically the Simscape [16] library. In this case, the communication is realized via Ethernet. During the evaluation, the platform monitors the faults effect not only on the controller but also on the mechanical part of the system - the stepper motor. To be able to do this, the platform utilizes a simulation with autonomous analysis of the motor behavior. It is also vital to choose a proper injection strategy as it has a significant impact on results quality.

IV. Experiments and Results

To test faults in the CPU and its memories during their operation, we use implementation of the MSP430 CPU for the FPGA, called the NEO430 [17]. The three original programs from our previous research [7] were extended with one new program, to isolate the effects of our SIFT methods. Further, we changed the naming of our previous programs, as these might be confusing in the context of our new analysis. Actual SIFT modifications made to the original code are shown in the Activity Diagram in Figure 1.

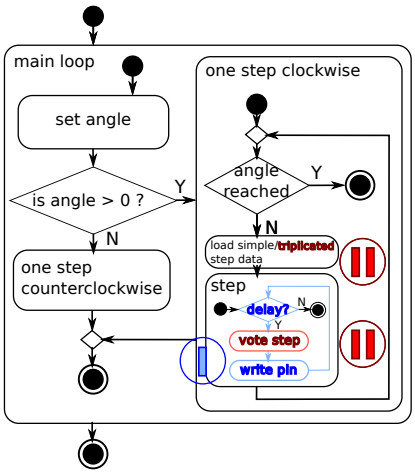


Figure 1: UML Activity diagram of the program with two modifications: ”I”) writes signals to output pins during the delay; ”II”) triples stored data and adds voting.

The program variants include: **1) Original Variant** (i.e. Variant O) – the unmodified control program; **2) Variant I** – it propagates signals to the output pins during the delay function execution; **3) newly added Variant II** – the motor excitation data are stored in three copies in the IMEM and are voted before their usage; **4) Variant I+II** – combines both the modifications I and II. For the implementation, we use the Xilinx Virtex 5 FPGA and synthesize the logic using the *Integrated Synthesis Environment* (ISE) 14.7. Again, we examined two injection strategies: 1) the single; and 2) the multiple fault injection.

A. Single Fault Injection Experiments

During the single fault experiment, one bit flip fault is injected before the CPU clock signal is enabled. For single experiments, injections into utilized bytes of IMEM and the CPU were examined independently. CPU injection is approximated through a bit flip in the occupied *Look-up Tables* (LUTs) of the CPU FPGA implementation. The faults were selected uniformly-at-random and 6,000 runs were performed for the CPU, while 2,000 runs for the IMEM. The results are shown in Table I. The first part of Table I classifies failures into ***Stuck*** – the motor stopped too early; ***Timeout*** – the motor did not

stop during the predefined interval of 220 s; and ***Mismatch*** – wrong data were observed on the output pins. The right part of Table I classifies the cases that achieved the correct angle although the electronic showed errors on its outputs. Although injection into the IMEM shows only slightly better results for the hardened programs I, II and I+II, the CPU injection shows the opposite trend. As can be observed, the hardening was apparently worsened by an unpredicted phenomenon.

TABLE I: The results of single injection experiments with failures classification; ”O”, ”I” and ”I+II” published in [7], extended with ”II”.

	Electronic Failure				*Mechanic OK (Out of Electronic Failed Runs)*			
	Total [%]	*Stuck* [%]	*Timeout* [%]	*Mis-match* [%]	*Total* [%]	*Stuck* [%]	*Timeout* [%]	*Mis-match* [%]
CPU ”O”	5.3	4.5	0.4	0.4	7.3	3.2	0.0	4.4
IMEM ”O”	36.7	15.6	14.0	7.0	20.3	1.4	1.0	18.0
CPU ”I”	5.9	5.1	0.5	0.3	8.4	3.9	0.7	3.9
IMEM ”I”	35.2	15.5	14.4	5.4	21.2	4.8	3.7	12.6
CPU ”II”	6.1	5.6	0.4	0.2	8.4	6.5	0.3	1.6
IMEM ”II”	34.6	16.0	11.9	6.7	24.2	3.0	2.6	18.5
CPU ”I+II”	6.2	5.5	0.5	0.2	10.4	5.1	0.3	2.4
IMEM ”I+II”	34.0	17.3	9.6	7.1	25.0	3.8	1.6	19.6

It is important to evaluate the mechanics behavior, too. In Figure 2, the final number of motor rotations for experiment runs in which the electronic failed is shown in a box plot chart. The desired 12.4 rotations is highlighted by the blue line.

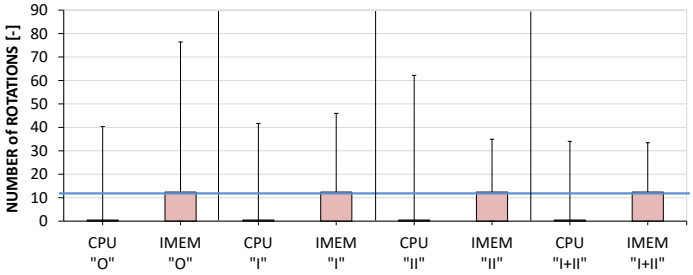


Figure 2: Box plot chart with final rotation for single injection; ”O”, ”I” and ”I+II” published in [7], extended with ”II”.

As can be observed, in most cases, the CPU injection caused that the motor did not start to rotate at all. For the IMEM injection, the hardened programs have the maximum angle slightly closer to the required value of 12.4 rotations.

B. Multiple Fault Injection Experiments

We made equivalent experiment with the multiple injection strategy. At the beginning of each run, the processor was started. After first 10 s, one bit flip was injected every 5 s. A run was active until the motor stopped on the required angle (usual duration of 80 s) or a timeout of 220 s was achieved. The faults were selected uniformly-at-random and 6,000 runs were held for the CPU target; 2,000 runs for the IMEM and 500 runs for the DMEM. These experiments also include injection into the DMEM, which is not meaningful for single injection strategy, as the DMEM contents is built during run time. The occupied DMEM was detected based on circa 1,000 DMEM read backs, before the injection experiments were started. Through the analysis of the read backs, we obtained the memory utilization map. Data are presented in Table II. The meaning of the columns is equivalent to the single experiments.

TABLE II: The results of multiple injection experiments with failures classification; ”O”, ”I” and ”I+II” published in [7], extended with ”II”

	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
IMEM ”O”	99.1	41.1	27.1	30.9	19.7	1.8	0.2	17.7
DMEM ”O”	91.8	9.0	15.4	67.4	13.1	0.0	0.0	13.1
CPU ”I”	70.1	13.6	30.1	27.0	16.3	2.2	3.2	10.9
IMEM ”I”	98.4	31.8	31.4	35.3	31.7	4.4	0.2	27.1
DMEM ”I”	92.8	11.6	15.6	65.6	13.8	0.0	0.0	13.8
CPU ”II”	68.7	16.0	40.3	12.4	14.8	3.4	3.9	7.6
IMEM ”II”	99.0	53.9	16.4	28.8	43.4	3.4	0.0	18.3
DMEM ”II”	99.2	60.6	2.6	36.0	6.9	0.0	0.0	6.9
CPU ”I+II”	89.0	19.6	23.4	46.0	12.1	1.9	1.0	9.2
IMEM ”I+II”	99.7	40.1	22.5	37.1	27.3	3.9	0.2	23.3
DMEM ”I+II”	95.4	34.2	3.0	58.2	17.2	0.0	0.0	17.2

As can be seen, generally the highest sensitivity has the IMEM. This is because it is often read, a change in its content alters the program behavior and the IMEM is never written to. This is why a fault in the IMEM has no possibility to eventually rewrite (i.e. repair) during the program run time. Also, as can be observed, we believe the higher DMEM occupancy worsens the results of the experiments with DMEM. We also believe that the higher utilization of the internal CPU bus for the ”I+II” program causes the deviation of the CPU ”I+II”, which has significantly worse results.

We also monitored mechanic part for the multiple injections. The final number of rotations for runs in which the electronic failed can be seen in box plot chart in Figure 3.

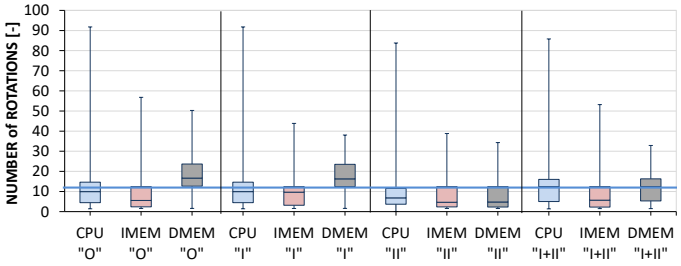


Figure 3: Box plot chart with final rotation for multiple injection; ”O”, ”I”, ”I+II” published in [7], extended with ”II”.

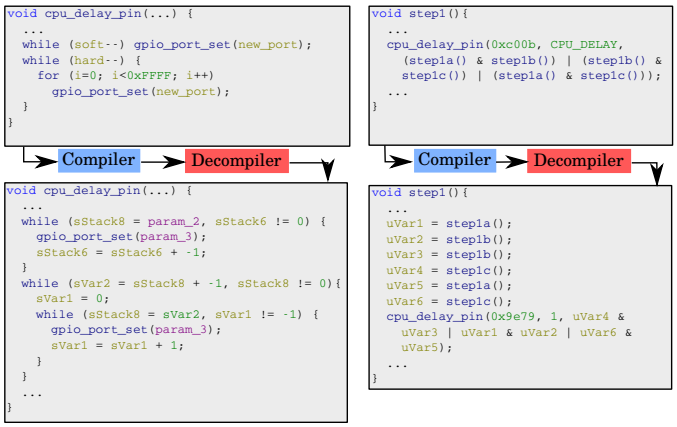
As can be observed, the rotation for ”I” compared to ”O” is very similar for the CPU and DMEM targets; for the IMEM, the ”I” is better. The ”II” is worse for all injection targets. However, the ”I+II” has the median closer to the expected rotation for the CPU and DMEM (i.e. better than the ”O”); for the IMEM, the rotation is very similar to the ”O”.

V. Analysis and Conclusions

Although the experimental results are interesting for assessment of faults impact on particular injection targets, the hardening itself did not bring significant improvements. From our point of view, it is very interesting to analyze this anomaly and publish the design error which caused this anomaly. In the following text, three hypothetical reasons are examined.

A. Compile-time Code Optimization

At first, we ensured our SIFT modifications remained in place after the code was compiled, although the lowest possible optimization level was selected. By using the Ghidra tool [18], we decompiled our binary programs for the NEO430. Code snippets of modifications "I" and "II" can be seen in Figure 4. As can be seen, the hardening remains in the binary program.



Code Modification I (Data are Continually Propagated during the Dealy Operation)

Code Modification II (Data are Stored Three Times in the IMEM)

Figure 4: Original vs. decompiled program code snippets for both of the modifications.

B. Accuracy of our DMEM Occupancy Detection

For our research, we developed the detector of DMEM address occupancy. This significantly accelerates the evaluation, as the average DMEM occupancy for our programs is 1.9 %. However, if a dynamic memory allocation is in place, the occupied addresses may be fragmented all over the address space. And for such cases, our detection method is not suitable. To evaluate the suitability of this method, we created heat maps of memory bytes occupancy. It is obvious that a high *temperature* on a small number of cells indicates a better suitability. On the contrary, a low *temperature* on a high number of memory addresses indicates the occupied cells are scattered. The heat maps can be seen in Figure 5.

It is obvious that the method is suitable for our programs, as the heat maps indicate occupancy of a few cells with a high probability of them being occupied.

C. CPU Internal Bus Traffic

The NEO430 is a 16-bit processor. It uses the internal host bus to communicate with its numerous components. Considerable amount of space is occupied by bus controllers

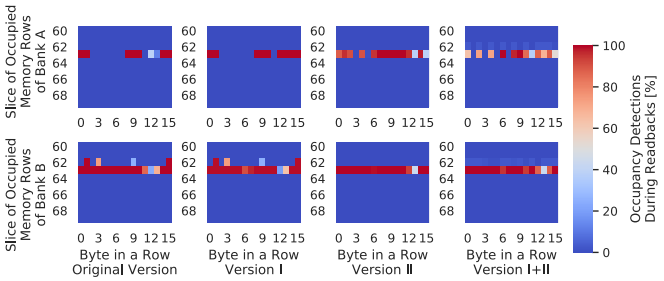


Figure 5: Heat maps of DMEM occupancy for each of the four programs.

of the components. It is, therefore, hypothetically possible that the added SIFT increased the transmission rate on the internal bus. This potentially enlarges the space for fault manifestation. For each program, we measured the number of read and write transactions. As the NEO430 distinguishes between both the bytes in a 16-bit write transaction, these were monitored independently (i.e. as the Byte 0 and Byte 1). We also monitored the amount of data transferred. The results are shown in Table III.

TABLE III: Bus traffic for each of the four program versions

	Program	***Original***	***Version I***	***Version II***	***Version I+II***
Transactions	*Read*	1556.6×10^6	1892.4×10^6	1529.5×10^6	1894.7×10^6
	Write Byte 0	29 143	287.1×10^6	29 768	287.1×10^6
	Write Byte 1	29 143	287.1×10^6	31 876	287.1×10^6
Data	*Read*	2968.9 MiB	3609.5 MiB	2917.3 MiB	3613.9 MiB
	Written	0.055 MiB	547.6 MiB	0.058 MiB	547.6 MiB

As can be observed, the high number of read transactions is caused by reading program instructions from the IMEM. Furthermore, the repeated propagation of results to the output pins (i.e. the modification "I") significantly increased the write transactions number for corresponding programs. These results indicate that, at least for the CPU experiments, the added hardness was partially cancelled by making the program more vulnerable due to increased bus traffic.

To conclude this paper, we found out that added hardness may improve resilience to faults. But also, the resilience may be instantly worsened by other factors, such as increased bus traffic. In our research we found out, that our hardening did not improve the resiliency to faults due to the increased bus traffic. This means that it is always important to consider also the parts of the system that are easily overlooked. And it is necessary to search for other critical points for the FT.

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] C. Salzmann, S. Govaerts, W. Halimi, and D. Gillet, "The smart device specification for remote labs," in *Proceedings of 2015 12th International Conference on Remote Engineering and Virtual Instrumentation (REV)*. IEEE, 2015, pp. 199–208.

[2] 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.

[3] C. Lee, Y. Chung, T. Shen, and K. Weng, "Development of electronic locks using gesture password of smartphone base on rsa algorithm," in *2017 International Conference on Applied System Innovation (ICASI)*, 2017, pp. 449–452.

[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] L. Pullum, *Software Fault Tolerance Techniques and Implementation*, ser. Artech House computing library. Artech House, 2001. [Online]. Available: https://books.google.cz/books?id=hqXvxsO5xz8C

[7] J. Lojda, R. Panek, J. Podivinsky, O. Cekan, M. Krcma, and Z. Kotasek, "Hardening of Smart Electronic Lock Software against Random and Deliberate Faults," in *Paper accepted for presentation at Digital System Design (DSD), 2020, 23th Euromicro Conference*. IEEE.

[8] R. S. Divya and M. Mathew, "Survey on various door lock access control mechanisms," in *2017 International Conference on Circuit ,Power and Computing Technologies (ICCPCT)*, 2017, pp. 1–3.

[9] 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.

[10] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 243–254.

[11] D. R. Avresky, S. J. Geoghegan, and Y. Varoglu, "Evaluation of software-implemented fault-tolerance (sift) approach in gracefully degradable multi-computer systems," *IEEE Transactions on Reliability*, vol. 55, no. 3, pp. 451–457, 2006.

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

[13] G. K. Verma and P. Tripathi, "A digital security system with door lock system using RFID technology," *International Journal of Computer Applications*, vol. 5, no. 11, pp. 6–8, 2010.

[14] 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.

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

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

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

[18] National Security Agency, "Ghidra - Software Reverse Engineering Framework," https://www.nsa.gov/resources/everyone/ghidra/, 2020, accessed: 2020-06-20.