Testing Embedded Software Through Fault Injection: Case Study on Smart Lock

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

***Abstract*—The growing chip-level integration results in a higher susceptibility to faults of today components. This also relates to commonly used storage memories. A charged particle causes bit flip and a program stored in such memory starts to behave differently from it was supposed to. Even worse, such bit flips can be induced also on purpose to tamper with a device. While the so-called smart devices are becoming still more popular these days, such failure or even tampering of them is very undesired. A smart electronic lock can serve as an example. This is why in this paper, we evaluate the consequences of such program corruption. We target smart lock operation on several computer architectures and show the results on our case study observing the change of the lock behavior. We present our Evaluation Environment that is able to connect with single-board computers and evaluation kits to test the SW behavior on them, which is done under the presence of faults in the tested SW. Our results indicate that the most sensitive part of a program is generally the loading of shared libraries. Problem in this process results in inability to load the program. Segmentation Fault and early termination of the program (e.g. problem in the logic of motor cycle counting) is also serious. The least problematic, according to our observations, is the syntactic error in the output data. In such cases, the motor driver ignores corrupted commands and the motor move is not smooth. Certain findings from the experimental part of this paper, can be generalized to other devices as well.**

***Keywords*—Electronic Lock, Stepper Motor, Software Fault Injection, Evaluation Environment, Linux, ARM, x64.**

I. Introduction

Electronics is everywhere around us and its usage is still growing. The increasing chip-level integration allows to implement complex systems into a very small area on a chip. But it also brings problems such as its higher susceptibility to failures. This is why certain systems are built with hardened reliability in mind. Reliable electronic designs are usually embedded in systems the reparation of which would be very expensive or even impossible (e.g. space satellites) or in systems where a failure would be very dangerous (e.g. medical equipment). However, the usage of reliable design also makes sense for a certain part of the consumer electronics sector.

The so-called smart devices are penetrating into our lives. One example of such a device is the so-called smart electronic lock. It is an improved version of an ordinary door lock. It benefits from the possibility to authenticate its user by other means, such as passwords or dual-tone multi-frequency signal [1]. The authentication can be even based on the recognition of a color [2], which then serves as a password. However, in order to achieve a high level of security, authentication methods should be combined [3]. Certain electronic locks can be connected to a local network [4]. A detailed description of such access control systems can be found in [5].

The consumer electronics is not usually considered to be safety-critical. Such electronics includes, for example, home assistants and other home automation systems and actuators, the popularity of which is rising sharply in recent times. The failure of such devices can be in certain circumstances potentially dangerous to human or animal health or eventually present a risk of financial losses. The failure can be introduced intentionally by an attacker or naturally because of aging of the device. Even for such types of systems, the consequences of their potential failure should be well analyzed and proper precautions done. Devices that perform decisions or guard the accessibility of certain places, such as transportation cabins or rooms in a building, must maintain certain level of reliability and security. For example, blocking of an electronic lock or an authentication terminal of an entrance during a fire, creates a very unnecessary obstacle. Research in the field of security of smart electronic locks is nowadays discussed in the literature. For example, the authors of [6] present security analysis of a commercially available smart electronic lock. In paper [7], other authors show security problems of another commercially available smart lock. It turns out that the back-end services suffered several vulnerabilities.

Any software can be analyzed on random and deliberate faults using the so-called fault injection, which is an artificial incorporation of faults into a software (e.g. a source code or the machine code). After a fault is injected, the altered behavior of the software can be observed and evaluated, which provides us with the insight to the software robustness. Specifics of the CPU architecture might contribute to this result. Fault injection into software is also a topic studied in the current literature. For example, the authors of [8] propose a novel technique called *G-SWFIT*, which uses operator emulation. The paper [9] presents a fault injection system called *Debug-based Dynamic Software Fault Injection System*. The authors of papers [10], [11] propose a method based on the QEMU simulator. In [12], the authors put software fault injection into the context of software certification. The authors of [13] demonstrate a software fault injection model based on the so-called software mutation. The authors of [14] propose a fault injection testing approach that accelerates the process of evaluation. Nonetheless, none of these addresses the consequences of fault injection into an advanced *Internet of Things* (IoT) device running on an embedded operating system. In our previous research, we focused on fault injection into CPU logic and memories of an embedded microcontroller on a smart lock with the usage of our *Field Programmable Gate Array* (FPGA)-based fault injection evaluation platform [15]. These were, however, targeting

978-1-6654-2057-0/21/$31.00 ©2021 IEEE

2021 IEEE 22nd Latin American Test Symposium (LATS) | 978-1-6654-2057-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/LATS53581.2021.9651770

the simplistic smart lock implementation on a microcontroller, i.e. without the involvement of the embedded operating system.

This paper contribution targets two main areas. 1) It is the contribution to a specific case of testing SW on embedded operating system on various computer architectures. We do consider the SW only (i.e. the operating system is not under the fault injection). We do this by utilizing our new Evaluation Environment, which connects to commercially available single-board computers or evaluation kits, to provide severe testing before the SW itself is considered robust. Our proof-of-concept implementation of the presented architecture shows that it is possible, with certain limitations, to test the software with such concept. 2) Another contribution is the case study targeting the motor controller program. This is a very unique part of the smart lock SW, which controls the lock motor driver. The results from the case study are, thus, unique to this component of the smart lock. For this reason, at the time of writing of this paper, it was not easy to find another such evaluation, that would allow to compare the results obtained in our case study.

This paper is organized as follows. Electronic lock description with the discussion of its reliability is presented in Section II. Evaluation Environment, which we developed to measure fault resiliency of a software, is presented in Section III. Evaluation of faults impacts artificially injected into our controller software is presented in Section IV. Section V highlights the results and concludes the paper.

II. Electronic Locks and Their Reliability

It is obvious that the hardware configuration among various electronic locks is very diverse. Generally, an ordinary smart electronic lock consists of the Control Module, which is usually realized as a computer with a CPU. There is also a Motor Module, which drives the needed mechanical force to eventually open or close the lock. Usually a stepper motor is used [16], [17]. The I/O Module then implements communication with the outer environment. It can interface the lock to a Local Area Network or a Cloud. Block diagram overview can be seen in Figure 1.

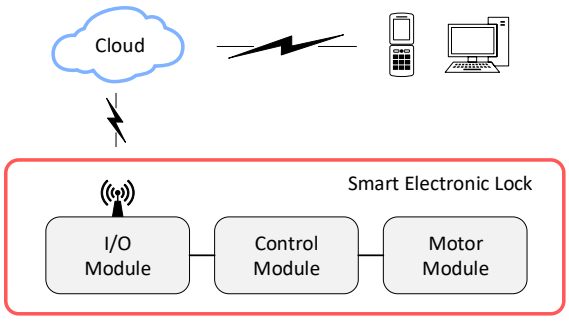

Figure 1: The block diagram of the smart electronic lock.

In the case of a smart electronic lock, which is connected to the Internet (becoming part of the IoT), the communication with the central server (Cloud) has to be mentioned. Based on valid credentials, the Cloud executes user commands to open or close the lock or other specific custom commands. [18] From the communication point of view of the smart lock with the surroundings, it is necessary to pay attention to its security.

Different manufacturers offer various locks with variant functionality on the market [19]. State-of-the-art locks can offer voice and image recognition, presence detection, remote firmware updates, and various real-time notifications. These locks can also be linked in a smart home and can be adapted automatically in different scenarios that may occur in the house (e.g. unlocked in the presence of fire, locked during the evening, etc.).

All the smart lock operations require sophisticated control, which is most often provided by a processor [20], [21], [22]. The processor controls all peripherals for communication with the outside world and operates the mechanical part of the lock itself. The processor is the brain of the entire smart lock, so attention must be paid to its security. It is necessary to ensure the correct evaluation of the unlocking request so that unauthorized access is not granted. As the results of independent tests show, only 4 of the 16 tested commercial smart locks are not vulnerable, which is very worrying. [18] For this reason, a huge attention should be devoted to secure all smart lock components.

Several attack types targeting the lock processor can be identified. Although the internal software and hardware components are hidden from the view of the user, two main different design approaches can be used for a smart lock main processor unit. The processor of the lock can be **I) a low-power microcontroller**, which can, perhaps, execute a real-time OS for such purposes. This is useful mainly for locks, utilizing a wireless connection to their environment and, thus, it is important to keep their power consumption on a very low level. Another approach is to contain **II) a more powerful *System-on-Chip* (SoC)** with the ability to execute a light version of an ordinary Linux kernel with the common shell applications. For such cases, the software can be programmed similarly as on an ordinary PC running a Linux OS. This approach, however, is not suitable for battery-powered smart locks.

Generally, the processor behavior can be changed in two different ways. The first way is to change **1) the processor logic**. This can be achieved as a result of a failure on the processor hardware logic level. Such failure can be a result of the hardware aging, which occurs naturally. It can also be evoked deliberately, as a consequence of an intentional tampering or an attack on the lock. The second case, in which the processor can start to behave differently than it was designed to, is **2) the modification of the program or data**.

One option is to **A) influence the processor program using the program input data**. This may represent a forgery of the credentials which will result in a successful unlock of the lock. The credential may also be leaked during the transmission over an unsecured channel. Due to the limited capabilities of some microprocessors, the latest encryption algorithms cannot always be used, therefore, the weak ciphers can be broken. Many devices are battery-powered for many years, so *Denial of Service* (DoS) [23] attacks can considerably deplete the battery due to intensive communication which shortens the lock life. The second option is to **B) change the program instructions and data** or induce logic faults into the hardware. Such faults can be induced naturally, which is dependent on the place of operation of the lock. These faults can be, however, also induced deliberately by the attacker's intention to manipulate internal data. It is very hard to detect such manipulation, as it does not require to disassemble the smart lock. It does not even require any mechanical contact

with the device. In our research, we focus on evaluation of such effects of faults on the controller program itself.

In our previous research papers [24], [15], [25], we were focusing towards the hardware logic robustness and memory tampering and evaluation on a simpler smart lock hardware, which can be classified as **I-1-B** and **I-2-B** in our classification from the previous three paragraphs. For our previous research, we used a different evaluation platform, that utilizes FPGAs to instantiate the component of a microcontroller and inject faults into such microcontroller. Our previous hardware, which was utilizing the NEO430 microcontroller, was featuring the *Software-Implemented Fault Tolerance* (SIFT) approach in the software. Such architecture is present on simpler locks. In this follow-up research, we target more advanced smart locks that incorporate an embedded operating system. Such devices are usually part of the building access system and are, thus, connected to an external, usually uninterruptible, power source. Such locks tend to utilize more powerful processor, running a stripped version of the Linux OS. This research paper, thus, examines part of the attacks classified as **II-2-B**. For targeting such attacks, we had to design a new Evaluation Environment, which is able to test software on various single-board computers and testing kits. We believe this solution for this test scenario precisely is more productive and universal, compared to porting the Linux on various CPU *Intellectual Property* (IP) cores for an FPGA. Because such porting has indeed been done for commercially available single-board computers and kits. The failing program can behave very differently under presence of faults, when executed on an embedded operating system. For example, in the case of Segmentation Fault error, the operating system cancels the target binary execution, which is not the case for the bare-metal implementation.

III. Evaluation Environment

For the purposes of our experimental testing and measurement, we propose our new Evaluation Environment. The Evaluation Environment focuses on testing programs running on the Linux OS. However, support for different operating systems can be considered, if such operating system offers possibility to 1) deliver and store a new binary program and 2) execute the program remotely and communicate with the program. In the case of Linux OS, the first is easily addressed by the *Secure File Copy* (SCP) and the second is addressed with the usage of the *Secure Shell* (SSH). The Environment can evaluate programs a) on the same computer as the Environment is executed; or b) remotely on a different computer (with possibly different architecture) using the SSH and SCP.

The obtained data are further analyzed and stored on the personal computer serving as the experiment supervisor. For such cases that allow to run the Evaluation Environment on the target computer architecture, it is also possible to run the tested program locally under a different operating system user. First, this is useful to prevent the failing program to interfere with the Evaluation Environment control program itself. Secondly, it also allows to limit the available resources (such as memory or storage write operations) to the tested program. The latter is also applicable to the remote binary execution scenario. To restrict the amount of Linux OS resources, the `ulimit` command can be used. The structure of the Evaluation Environment can be seen in Figure 2 for both the local and the remote binary program executions.

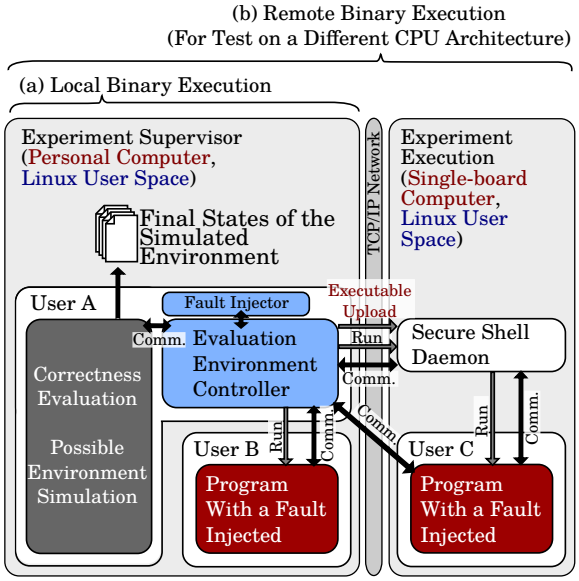

Figure 2: The architecture of the Evaluation Environment for execution and testing of programs running (a) on the same computer as the Evaluation Environment; (b) remotely on another computer utilizing (possibly) different architecture.

A. Fault Injection Software Evaluation

Several runs are needed to evaluate one program. The number of runs is usually based on the size of the tested program. The program evaluation looks as follows:

1) a program is compiled for the target architecture,
2) the Evaluation Controller is started on a PC,
3) repeat until the number of runs is achieved:
   a) one bit is flipped in a copy of the binary executable,
   b) the binary is copied to the local or remote storage,
   c) OS resource limits are set and the binary is executed,
   d) the outputs are time stamped, stored and analyzed,
   e) if no data are observed for a certain period of time, the executable is stopped; if the timeout limit is reached, the executable is forced to stop,
   f) the stored results are further analyzed.

B. Testing in the Natural Environment

For certain scenarios, the tested program is actually a controller logic. For such scenarios, 1) the program obtains data from the environment simulation and 2) the simulation is driven through the data obtained form the running program. For certain types of systems, only the second is applicable. In such cases the program does not sense any data back from the simulation and it only drives the simulation. In these cases, the data may be stored and the simulation can be performed later, which is useful if the simulation cannot run in real time.

C. Prolonging Service Time of the Test Target

It is important to note that, for the remote execution, the binary must be compiled for the target architecture. For the remote execution on computers utilizing a flash storage, it is advisable to create the so-called *RAM disk* on the target computer to store the modified binary. This prolongs the service life of the target computer, as the number of experimental runs is usually at the magnitude of tens of thousands and the continuous rewriting of the flash is not desirable in such case.

IV. Experiments and Results

The aim of our experimentation is to find the most fault-sensitive parts of a controller program. This experimentation is performed on three different CPU architectures to possibly confirm or exclude influences of the CPU architecture on the result. Our experiments are performed on a generic example of a smart electronic lock. This is because the smart lock belongs to the category of such IoT devices that represent certain risk in a case of their failure. Our experiments were held with the usage of the Evaluation Environment, which was described in Section III. For the experiments we used our own electronic lock, which is composed of an electronic controller and a stepper motor. The electronic controller is implemented on a Linux single-board computer, which communicates through a serial line with the motor driver. Our intention is to test the behavior of a smart lock motor controller program under the presence of bit flips (i.e. injected faults) in the program binary code. The aim of the particular tested software is to control a stepper motor driver based on the required angle. The driven motor is then supposed to open or close the lock. As the motor driver communicates through the serial line, the motor controller program, thus, commands the driver through its *standard output* (stdout). The system structure, from which the Motor Module is simulated in software, can be seen in Figure 3.

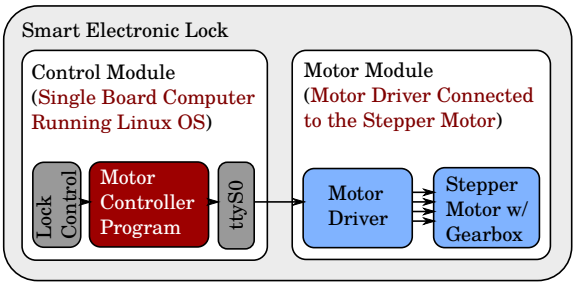

Figure 3: The block diagram of the smart electronic lock; the Single-board Computer is actual hardware part and the Motor Driver, Stepper Motor and the Gearbox are simulated on a PC.

In these experiments, security and safety of this motor controller program is evaluated through artificial fault injection of single bit flips into the whole binary program file. In our study, we assume a failing data storage on which the controller program is stored. This could be because of a natural aging of the components or intentionally, as the memory could be manipulated by an attacker, which has a possibility to *blindly* inject faults into the binary control program inside the electronic lock. For our tests, we assume that the potential attacker does not have a deep knowledge on the electronics layout and the SW structure, which would allow the attacker to target faults towards specific parts of the program. Thus, we also assume that these memory errors are distributed uniformly-at-random throughout the whole controller program. Thus, during one run, always one uniform-at-random bit flip is injected into the binary to emulate a hard corruption before the execution. Our tests focus on the program, while the Linux kernel is not included in our tests. Several runs are needed to evaluate one program. The number of runs is dependent on the program size. Based on our previous experiences described in paper [26], we test circa 10% of bits available for the fault injection. For the stepper motor simulation, we use very accurate model from the Simscape library of Matlab Simulink software [27].

A. Target Architecture Influences

The experiments were held on three different CPU architectures: **1) ARMv6Z** from the ARM11 family, 700MHz single-core ARM1176JZF-S; **2) ARMv7-A** from the Cortex-A family, 1.3GHz dual-core ARM Cortex-A7; and, for the reference purposes, **3) x64** (a.k.a. x86_64) 2.4GHz 6-core Intel Xeon E5-2620v3 64-bit server CPU, to possibly compare the results of the two embedded architectures to an ordinary 64-bit PC CPU. The controller program was compiled for each architecture. For each compiled program, $8\,000$ runs were performed. Each run was evaluated according to the correctness of its output data.

There are two classes based on the correctness of the output data. The class ***"OK"*** contains the cases in which the injected fault was not observable on the program output data, i.e. commands for the motor driver. The class ***"Failed"*** contains the opposite cases. We decided to prolong the execution of the program to $80\,\mathrm{s}$, in order to increase the time to observe the program behavior. The reference execution, thus, lasted $80\,\mathrm{s}$, during which the program was instructed to rotate the motor 12.4 times. The results with the percentage representations of these cases are displayed in Table I. An error in the output data does not necessarily indicate a problem in the operation of the lock. Because for the end user, the functionality is important. The fact that a controller received partially scrambled command is very unimportant in the case of an attack or a natural degradation (e.g. for the smart lock, in the case of emergency). This is why we analyze also the motor module behavior. Because of this, the table also contains minimal, average and maximal rotation angles for each category. A sign in front of the number of rotations represents the direction of the rotation. If a run achieves the required 12.4 rotations with a slight deviation of 0.2 rotations, such run is considered successful lock or unlock. If the rotation angle is smaller, the lock state does not change for an external observer, i.e. stays locked or unlocked. Preventing the door to unlock is potentially even worse than refusing to lock the door. A locked person in a building on fire is a catastrophic failure from the safety point of view. If the final angle of the motor is higher that the required 12.4 rotations, the mechanics or the motor are possibly damaged and are likely to fail during their future requests.

As can be further observed in Table I, the failed percentages among the CPU architectures are very similar. It means that the architecture did not significantly impact the resulting reliability of the program. Also, the minimal angle of failed runs for the ARMv6Z architecture is significantly smaller, compared to the other two architectures. This behavior was the result of injecting fault into the section of the program that chooses the direction of the motor rotation (i.e. for locking or unlocking). Obviously, this precise target was not impacted for the other two architectures.

B. Failure Types

According to further analysis of the received data, five categories of failure reasons were identified. These include: **1)** ***"Lib. Error"***, containing the runs that show problems with load or relocation of shared libraries; and **2)** ***"Seg. Fault"*** including all executions that showed signs of memory access violation. The category **3)** ***"Early Term."*** contains all cases that stopped their execution before the desired result was achieved (i.e. before the motor achieved the desired position). This can

TABLE I: Behavior of the Program Under the Fault Injection

	ARMv6Z				ARMv7-A				x64			
	Cases	Rotations [-]			Cases	Rotations [-]			Cases	Rotations [-]		
	[%]	min.	avg.	max.	[%]	min.	avg.	max.	[%]	min.	avg.	max.
All Runs	100.00	-35.37	11.11	35.53	100.00	-0.09	10.87	35.68	100.00	-0.06	10.75	37.87
OK	88.16	12.40	12.40	12.40	86.05	12.40	12.40	12.40	85.56	12.40	12.40	12.40
Failed	11.83	-35.37	1.52	35.53	13.95	-0.09	1.42	35.68	14.43	-0.06	0.96	37.87

be caused, among others, by a problem in the cycle counting of the program or by a Segmentation Fault during the data output phase of the execution. **4) *"Syntax Err."*** contains the cases that violated the format of commands parsed by the motor driver. In such cases, the motor driver simply ignored such commands. The last category **5) *"Timeout"*** contained cases that continued their execution beyond the supplied time, which was 220 s. It is important to note, that each failed run can belong to multiple categories, for example, if the run produced corrupted format of output data and then stopped on Segmentation Fault.

The percentage representations of these categories, based on all the 8 000 runs, are displayed in Table II. We believe, that the reason for the higher representation of Segmentation Fault errors on the x64 might be the more complex instruction set of the x64. Alternatively, it might relate to the principle of detection of the *illegal* memory access or, generally, the principle of the memory management. This is not a negative finding, as the reference x64 is not meant to be used in ordinary embedded systems, in opposition to the widely-used ARM architectures. Also, among the architectures, growth in Early Termination is observed. This is because the Segmentation Fault sometimes occurs during the run, not just immediately after the execution starts. Also a lowering trend of Timeouts is observed in the same order.

TABLE II: Categories of Failure Types of the Program and Their Representation among All Runs

Cases [%]	Lib. Error	Seg. Fault	Early Term.	Syntax Err.	Timeout
ARMv6Z	1.13	7.85	10.58	0.90	0.25
ARMv7-A	1.23	8.75	12.61	1.03	0.14
x64	1.13	10.91	13.43	0.88	0.09

In Figure 4, statistical box plot charts are displayed for particular categories of failures from the previous table. As can be seen on the chart, the very critical part of the program is the part dealing with shared libraries selection and loading. If such section is corrupted, the program does not start at all, thus, as can be seen, the motor angle stays at zero. This is also very important finding that affects other embedded software as well. This problem, obviously, will not be limited to smart locks and even not to the IoT segment itself. This is because a large amount of programs utilize shared libraries (except, for example, statically linked binaries). The Segmentation Fault occurs also usually at the beginning of the execution, causing the motor not to start at all in most cases. Also Early Termination during the program execution occurs usually at the beginning of the execution. On the other side, runs producing syntactically wrong data usually achieve the desired rotation angle. This is because, usually, the Syntax Error is contained within one driver output configuration. Although the motor driver ignores corrupted commands, the motor rotation further continues. Its move is not, however, smooth. This finding might be applicable to other actuator systems that are driven in steps and are able to perform the required move even when certain amount of steps is omitted. As can be also observed, the Timeout can be caused by exceeding the required angle or by slowing the output data. In the second case, the motor eventually reaches desired angle, however, not in time. For certain devices, this might be a problem. To some extent, the slowing is however acceptable. For example, a smart lock unlocked with one-second delay is not much of a problem.

V. Conclusions

In this paper, we proposed a new Evaluation Environment for testing and measurement of software resiliency against fault injection. The environment was used to evaluate properties of our software controller, running on an electronic smart lock. In the evaluation, we utilized the ability to execute the tested program remotely, and thus, we tested the program on multiple CPU architectures. The results show, that most sensitive part is the loading of shared libraries. Such problem in this section of the program causes inability to load. This is a very important problem that is not limited to smart locks nor IoT devices. If a program is corrupted in such way, that the embedded operating system does not load the program properly, then this effectively results in a permanent malfunction of the device. Such problem is the most critical one.

Segmentation Fault and early termination of the program (e.g. problem in the logic of motor cycle counting) is also serious. However, in certain fault injection cases, the program exceeded the required angle. There were also the cases, in which the motor rotated too slowly, and thus, the required angle was not reached in time. Such problem is dependent on the target application. For example, if the smart lock unlocks with a one-second delay, it does not pose a risk on a human life. On the opposite side, if the motor is instructed to rotate significantly faster, is could skip the necessary steps. This might result in a partially-moved lock latch, effectively blocking the door. The least problematic, according to our observations, is the syntactic error in the output data. In such cases, the motor driver ignores corrupted commands and the motor continues its move. Nevertheless, the move is not smooth. The motor then usually reaches its desired position. This finding is applicable to other actuator devices as well. Such device must be driven in steps and must be able to perform the required move even when certain amount of steps is omitted.

As a future work, countermeasures to the tested program could be added and the resulting behavior could be studied. It would be interesting to observe the effectiveness of the countermeasures on the various program failure classes. The most challenging problem, according to our findings, will be the hardening of the library section of the program.

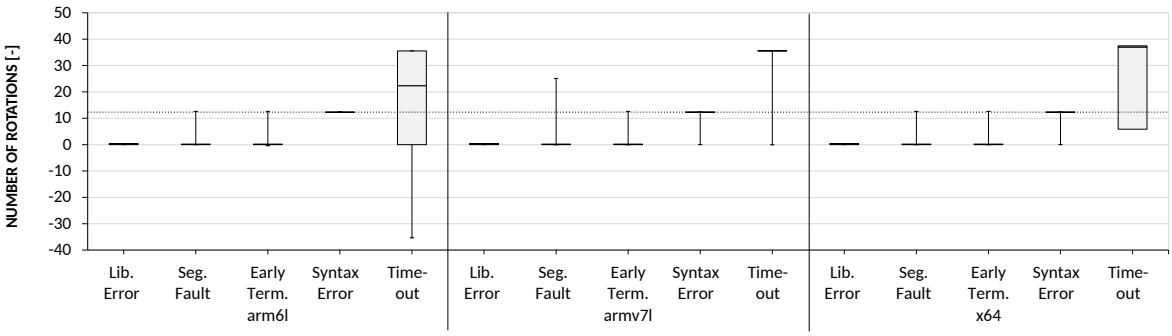

Figure 4: Final rotations per each category of failure type on a box plot chart.

ACKNOWLEDGEMENTS

This work was supported by 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] H. A. Rangkuti and J. W. Simatupang, "Security lock with dtmf polyphonic tone sensor," in *2015 International Conference on Automation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT)*, 2015, pp. 119–122.

[2] M. D. N. Chowdhury, M. K. Jahan, S. S. Karmokar, and S. Mahmud, "Color lock: 16 bit digital color based security system," in *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 2014, pp. 1–5.

[3] M. Mathew and R. S. Divya, "Super secure door lock system for critical zones," in *2017 International Conference on Networks Advances in Computational Technologies (NetACT)*, 2017, pp. 242–245.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[9] Y. Zhang, B. Liu, and Q. Zhou, "A dynamic software binary fault injection system for real-time embedded software," in *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011, pp. 676–680.

[10] J. Xu and P. Xu, "The research of memory fault simulation and fault injection method for bit software test," in *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, 2012, pp. 718–722.

[11] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "Qemu-based fault injection for a system-level analysis of software countermeasures against fault attacks," in *2015 Euromicro Conference on Digital System Design*, 2015, pp. 530–533.

[12] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[13] L. Yin, Y. Ri-huang, B. Jian-wei, and Y. Chun-hui, "Research on a software fault injection model based on program mutation," in *2015 2nd International Conference on Information Science and Control Engineering*, 2015, pp. 419–423.

[14] E. Cioroaica, J. Jahić, T. Kuhn, C. Peper, D. Uecker, C. Dropmann, P. Munk, A. Rakshith, and E. Thaden, "Accelerated simulated fault injection testing," in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 228–233.

[15] 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 *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, pp. 680–683.

[16] 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.

[17] Z. Fonea, "Electronic lock system," Nov. 14 2000, uS Patent 6,147,622.

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

[19] J. R. Delaney, "The Best Smart Locks of 2018," https://www.pcmag.com/article/344336/the-best-smart-locks, 2018, accessed: 2019-04-10.

[20] D. Han, H. Kim, and J. Jang, "Blockchain based smart door lock system," in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, 2017, pp. 1165–1167.

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

[22] M. Presso, D. Scafati, J. Marone, and E. Todorovich, "Design of a smart lock on the galileo board," in *2017 Eight Argentine Symposium and Conference on Embedded Systems (CASE)*, 2017, pp. 1–6.

[23] L. Prudente, E. Aguirre, A. F. M. Hdez, and R. J. García, "Dos attacks flood techniques," *International Journal of Combinatorial Optimization Problems and Informatics*, vol. 3, no. 2, pp. 3–13, 2012.

[24] J. Podivinsky, J. Lojda, R. Panek, O. Cekan, M. Krcma, and Z. Kotasek, "Evaluation Platform for Testing Fault Tolerance: Testing Reliability of Smart Electronic Locks," in *2020 IEEE 11th Latin American Symposium on Circuits and Systems (LASCAS)*, 2020, pp. 1–4.

[25] J. Lojda, R. Panek, J. Podivinsky, O. Cekan, M. Krcma, and Z. Kotasek, "Analysis of Software-Implemented Fault Tolerance: Case Study on Smart Lock," in *2020 IEEE East-West Design Test Symposium (EWDTS)*, 2020, pp. 1–5.

[26] J. Lojda, J. Podivinsky, Z. Kotasek, and M. Krcma, "Majority Type and Redundancy Level Influences on Redundant Data Types Approach for HLS," in *2018 16th Biennial Baltic Electronics Conference (BEC)*, Oct 2018, pp. 1–4.

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