Case study on combined validation of safety & security requirements

Michal Sojka, Michal Kreč, Zdeněk Hanzálek
Czech Technical University in Prague,
Faculty of Electrical Engineering
Technická 2, 121 35 Prague 6, Czech Republic
Email: {sojkam1,krecmich,hanzalek}@fel.cvut.cz

***Abstract*—In the automotive domain, it is of paramount importance to ensure safety, and recently also security, of the developed products. In many cases safety and security are handled separately by independent teams. In this paper we deal with testing and validation of safety- and security-related properties of control software in the AUTOSAR architecture and show that the strict separation of those two activities is not necessary and that combining them can bring economic benefits. We demonstrate that by developing software-in-the-loop and hardware-in-the-loop testbeds and use them for both safety- and security-related testing activities. We evaluate a prototype of electric motors control software, that is currently under development by Infineon Technologies, and perform a number of tests to verify correct functionality of implemented safety measures even under the presence of attacks. The motor control software is integrated with a message authentication protocol on CAN bus. The results show, that apart from few minor problems, the implemented safety measures function correctly.**

I. Introduction

The boom of electronic systems in cars brought also the need to guarantee their safety, i.e. to reduce the probability of their failure and severity of damage that can be caused by it as much as possible. This led to development of safety standards like IEC 61508 and the automotive adaptation ISO 26262 [1]. These safety standards provide guidelines and procedures for development of safe electronic systems and respective software. All these procedures involve extensive testing in many stages of the development process.

Later, the continuing integration of various electronic systems present in modern cars resulted in the need to access information from most (if not all) on-board systems in various subsystems in the car or even at remote locations. This trend resulted in the whole new area of concern – security. It is no longer sufficient to prevent system faults and minimize their impact, but now we need to defend against intentional attacks or malicious efforts to "upgrade" the control software [2]. Security is, of course, not a completely new field and there already exist security standards (e.g. ISO 15408 [3]), but they were developed mainly for traditional IT applications, not for the automotive domain. Therefore, they lack the necessary degree of consideration of safety features.

Because safety is already well-known and well-established in the automotive industry but security is something new and imported from outside, the safety and security issues are usually solved separately. Separate teams are usually responsible for safety and security concerns, and testing of safety and security measures is usually carried as separate processes. This leads to increased development costs, prolonged development time and possibly even to conflicts between implemented safety and security measures, which may result in decrease in their efficiency or even thwart their function altogether.

In this work we present a case study showing that such separation of safety and security is not necessary, at least in the testing and validation phases. With the help of Matlab/Simulink, we developed a reusable Linux-based testbeds for software- and hardware-in-the-loop simulations [4] and show that they can be used to validate both safety- and security-related requirements. The item to be tested is a prototype of a complex device driver for the AUTOSAR architecture named *eMotor*, currently being developed by Infineon Technologies. It is a software module for controlling several types of electric motors and it is supposed to be run on Infineon's TriCore TC1798 processor. For the purpose of security testing we implemented[1] a message authenticated protocol for the Controller Area Network (CAN), proposed by Volkswagen, called MaCAN [5]. We integrated MaCAN with the eMotor prototype and used this external interface to execute attacks on the eMotor software while observing eMotor behavior.

This paper is structured as follows. In Section II we give an overview of Infineon's eMotor driver. Section III describes software- and hardware-in-the-loop testbeds developed for this case study. The experiments run on those testbeds are described and their results are given in Section IV. We conclude with Section V.

II. eMotor driver prototype

The eMotor driver is a software module – more specifically a complex device driver – for the AUTOSAR software architecture [6], currently being developed by Infineon Technologies. It is meant to control electric Permanent Magnet Synchronous Motors (PMSM) and Brushless Direct Current motors (BLDC). It implements two control algorithms: Field Oriented Control (FOC) for PMSMs and Block Commutation (BC) for BLDC motors, providing torque control using PI controller(s). Motors are controlled by generating a PWM signal for the inverter. BLDC motors use 1-phase PWM signal and PMSM use 3-phase PWM signal. In this work, we consider only PMSM motors and therefore only the FOC algorithm.

[1] https://github.com/CTU-IIG/macan

The eMotor requires measurement of electrical current in the controlled motor and it supports multiple methods of implementing this. It also supports several sensor types for determining the position of the motor shaft, including a sensorless option. The eMotor driver is also equipped with a prototype implementation of several safety measures that should detect hazardous states. For example, the generated PWM signal is also read back in order to validate the function of the PWM unit. More details can be found in Section II-B.

The eMotor has an interrupt based design, where the actual control algorithm is executed in a hardware interrupt handler, which is invoked at the end of the analog-to-digital conversion (ADC) for current measurement [7]. It is designed to run on the 32-bit TriCore TC1798 microcontroller [8] developed by Infineon for automotive safety applications. The eMotor driver is configured with AUTOSAR configuration management tool EB tresos Studio from ElectroBit[2].

A. Current sensor configurations

The eMotor driver supports following current measurement configurations:

1) Two phase parallel: In this configuration, currents of two phases (i_a and i_b) are measured using two ADCs that measure simultaneously. The current of the third phase i_c is calculated from equation $i_a + i_b + i_c = 0$.

2) Two phase sequential: In this configuration two phases are measured using only one ADC. One phase is measured, then second one and then the first one again, averaging the result, to estimate the value of the first current at the time of the second phase measurement. Then the equation $i_a + i_b + i_c = 0$ is used to calculate the current through the third phase.

3) Three phase measure: In this configuration all three phases are measured with two parallel ADCs using the combination of the previous two configurations. One ADC is used to measure i_a, then both ADCs measure i_b and i_c and finally i_a is measured again and averaged with the first measurement. The equation $i_a + i_b + i_c = 0$ can be than used to check for anomalous behavior.

4) DC link measurement: This configuration is used with the Block Commutation algorithm.

B. Safety measures

The eMotor driver prototype implements four mechanisms to detect anomalous, potentially dangerous, situations. They serve only for detection of anomalous situation, responses to the errors were not yet implemented in the version available to us. These features are optional and can be turned off via compile-time configuration. The four safety measures are:

1) Current validation: This safety measure validates the plausibility of measured phase currents by checking their sum to be zero (or close to zero to account for measurement and numerical inaccuracy). If the sum is greater than a predefined limit an error is reported. This safety measure only works with three phase current measurement, because in other modes the third current is calculated from the other two, so the sum is always zero.

Besides the above check, there is an option to set upper and lower limits for each phase current, and if the current stays outside of these bounds for a given period of time a different error is reported. Note that this check cannot be disabled via configuration.

2) Position validation: When a sensor is used for position acquisition the mathematical motor model implemented in eMotor driver for sensorless mode can be used to detect anomalous behavior. If enabled, the motor shaft position is both read by the sensor and estimated by the internal model. If these two values differ more than a predefined threshold an error is reported.

3) PWM diagnostics: This safety measure diagnoses the functionality of the PWM generating hardware and/or connecting wiring by feeding the generated PWM signal back to the CPU and measuring its duty cycle with on-chip timer modules. If the measured duty cycle differs from the expected one by more than predefined threshold an error is reported.

4) Memory validation: With this feature enabled a complete copy of eMotor configuration parameters (motor and controller parameters, safety thresholds, etc.) is stored in the memory and before every major operation (measurements, control action calculation, etc.) the working and backup copies are compared by calculating a checksum with CRC32 algorithm and in case of a mismatch an error is reported. This enables detecting HW memory faults as well as unauthorized manipulation of eMotor parameters.

III. Developed testbeds

This section describes the two testbeds developed for testing of eMotor driver prototype – the software- and hardware-in-the-loop testbeds. Both testbeds are developed in Matlab/Simulink[3], which is a tool for multidomain simulation and model-based design heavily used in the automotive domain.

The SW-in-the-loop testbed runs completely on a PC, simulating both the controlled motor and the eMotor software. It is used for testing the functionality of eMotor software, the implemented safety functions and for integration testing of the said functions. This testbed does not contain the MaCAN (authenticated CAN communication) module.

The HW-in-the-loop testbed utilizes the actual hardware that the eMotor is intended to run on – the TriCore TC1798 microcontroller. PC-based simulation is used only to simulate the controlled motor. This testbed contains the MaCAN module and is in fact used for integration testing of said module, as well as for SW/HW interaction and subsystem functionality testing.

We use R2012b version of Matlab/Simulink for all Simulink models used in this work [9].

A. Software-in-the-loop testbed

The SW-in-the-loop testbed is a Simulink model depicted in Figure 1. It contains a dynamic model of the PMSM motor, a block with the eMotor code and blocks that simulate various faults. It is designed to work with all current measurements

[2]http://automotive.elektrobit.com/ecu/eb-tresos-studio

[3]http://www.mathworks.com/products/simulink/

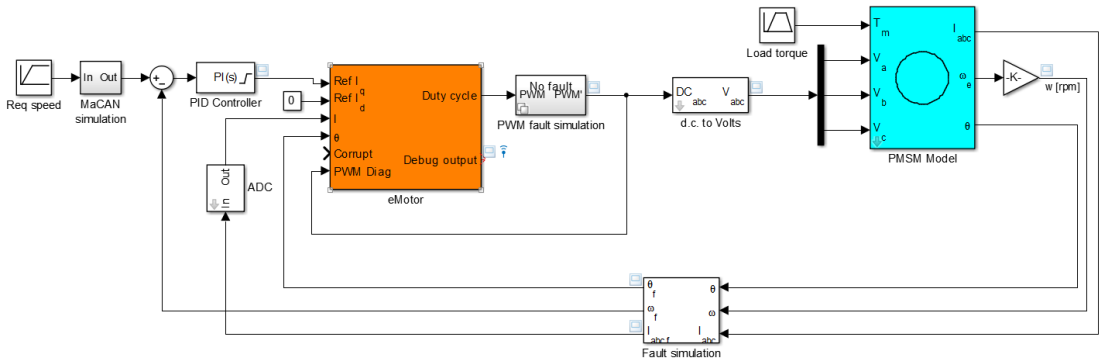


Figure 1: Software-in-the-loop testbed (Simulink model)

methods except DC link (we focus on FOC rather than on BC) and with all position sensors. The sensorless mode is currently not supported. The eMotor block reads shaft angle and 3-phase currents from the motor model and outputs a vector of three PWM duty cycle values, which is converted to voltage in order to act as an input for the motor model. There is also a standard Simulink PI controller block, serving as speed controller. In other words, it controls the eMotor's torque input in order to follow reference speed.

1) eMotor block: The eMotor block shown in Figure 1 is implemented as a C MEX S-function. This means that the block is implemented in C language rather than composed from other Simulink blocks. It comprises of slightly modified Infineon's eMotor driver code and Simulink interface code. The modifications reside in using Simulink interfaces rather than TriCore peripheral modules to read inputs and write outputs. Moreover, the eMotor code is invoked from S-function callback functions rather than by hardware interrupts. This allows us to simulate as much of eMotor behavior as possible. The sampling frequency (in simulation time) of the eMotor block was set to 20 kHz – the recommended value for eMotor control frequency.

2) PMSM motor model: To simulate the PMSM motor, we used a Simulink model obtained from Matlab Central[4]. The model implements differential equations describing a simplified PMSM in the *d-q* plane as well as Park's and inverse Park's transformations for voltages and currents [10], [11].

3) Fault simulation: The testbed naturally supports simulation of sensor and/or control faults. This is accomplished by variant subsystem blocks inserted between the motor model and the eMotor block. These subsystems allow for different behavior based on the value of a control variable. We introduced several *fault locations* in the Simulink model where faults can be simulated. The locations are: current measurement, position measurement and PWM diagnosis. At each location we are able to simulate several *fault types* (e.g. additive or multiplicative) and for each fault location there is a vector of start and stop times (e.g. the fault can be active at times 1–5 and 7–9 s). Additionally, for some fault types, their magnitude can be altered. Different fault types are described below.

Faults types common to all fault locations:

1) **No fault** – no fault is introduced in this setting.
2) **Additive error** – a constant amount, specified by the fault magnitude is added to the signal.
3) **Multiplicative error** – the signal is multiplied by amount specified by the magnitude.

Fault types specific to current measurement:

4) **Short circuit** – the current signal is set to zero.

Fault types specific to position measurement:

5) **Stuck** – the position is held at the value it had at start of the fault.
6) **Slipping** – maximum rate at which can the position change is limited to the fault magnitude.

Fault types specific to PWM diagnosis:

7) **Wire break** – the generated PWM duty cycle is set to 0.
8) **Min** – the PWM duty cycle cannot drop below fault magnitude.
9) **Max** – the PWM duty cycle cannot rise above fault magnitude.

B. Hardware-in-the-loop testbed

In a hardware-in-the-loop simulation, the tested software runs in real time on real hardware. In our case the eMotor driver runs on Infineon's development board called TriBoard[5].

Our HW-in-the-loop testbed is depicted in Figure 2. The TriBoard is connected to a PC via a PCI-based Humusoft MF624 I/O card[6]. PMSM motor model and fault simulation are run on the PC and the I/O card is used to connect the simulated motor with the board.

For real-time simulation, we use Simulink-provided external simulation mode. In this mode C code is generated from

[4]http://www.mathworks.com/matlabcentral/fileexchange/38804-pmsm-simulation.

[5]Infineon Order Nr. KIT_TC1798_SK

[6]http://www.humusoft.cz/produkty/datacq/mf624/

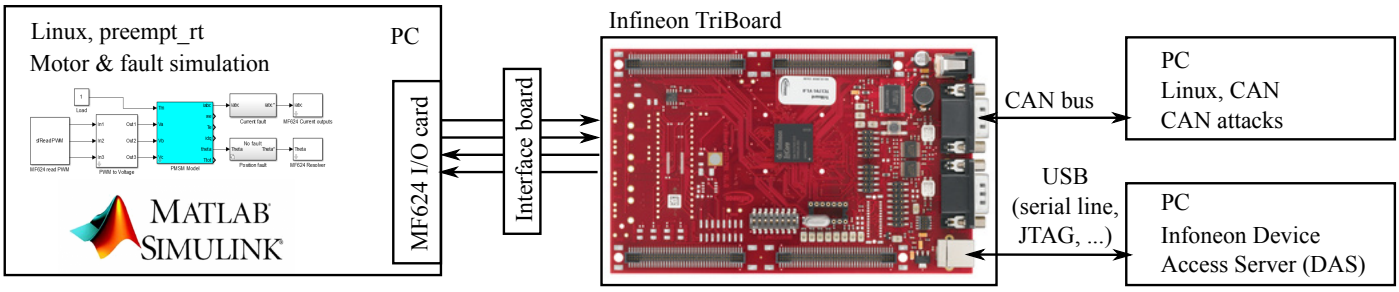


Figure 2: Block diagram of the hardware-in-the-loop testbed

the Simulink model, the code is compiled and run in real-time on the host system. Simulink provides only the user interface, i.e. it is possible to tweak model parameters at run time or to see graphs of various signals.

Since the eMotor driver runs with 20 kHz sampling frequency, it is necessary for the simulation environment to run also at this frequency. To meet this stringent timing requirements without buying an expensive hardware, we developed a custom code generation target [12] for Embedded Coder toolbox that generates code optimized for Linux with "preempt_rt patches"[7]. This setup allows us to run the simulation without a deadline miss at 20 kHz on an ordinary PC computer.

The computer with a CAN interface card was added later, in order to test the influence of CAN communication and message authentication on the eMotor's safety and time properties. This computer was used only in tests covered in Section IV-B4.

Some components of the testbed are described in more detail below.

1) Humusoft MF624 I/O card: The MF624 is a PCI expansion card designed for interconnection of a PC and real world signals. It features 8 channel 14 bit A/D converter, 8 channel 14 bit D/A converter, 8 bit digital input port, 8 bit digital output port, 4 quadrature encoder inputs and 5 timers/counters as well as fully 32 bit architecture [13]. Only D/A converter, digital input and timers are used in our HW-in-the-loop testbed. For that we developed Simulink blocks that access this card under Linux via the user-space I/O (UIO) driver[8].

2) Simulink model: The Simulink model used for HW-in-the-loop simulation contains the model of the motor described in Section III-A2, MF624 blocks mentioned above, unit conversion blocks, simulation of the resolver sensor and fault simulation blocks. The fault simulation was modified to allow fault control from the TriBoard by means of a logical signal that is controlled by eMotor software and read by MF624's digital input. A predefined fault is active whenever this signal is active. This allows precise time measurement of fault detection response times, because the precise time of fault initiation can be saved alongside the time measurements. Faults used here have the same types as those used in SW-in-the-loop testbed (see Section III-A) with the exception of PWM fault which is not implemented, because it would require additional hardware.

3) TriCore application: The eMotor demo application supplied by Infineon was taken as a basis for the testing program. This demo application uses the eMotor driver and extends it with a PI speed controller and a simple command line interface accessed over virtual serial port. The user interface allows for switching the motor control on and off, setting the reference speed, reading from position sensors and calibrating them. We extended the application to allow for direct PWM control, safety measures diagnostics (see Section II-B), measurement of eMotor execution time, fault control and transfer of measured values to a PC.

4) CAN bus & message authentication: The eMotor driver was enhanced with Infineon's AUTOSAR compatible CAN driver [14] for the purposes of testing the effects of CAN communication and possible attacks over CAN on the eMotor software.

Additionally, the message authentication protocol for CAN (MaCAN [5]) was implemented utilizing TriCore's Secure Hardware Extension (SHE) and integrated with the eMotor. MaCAN's *time server* and *key server* run on another Linux PC with the CAN interface card [15], together with simple application allowing secure motor control by means of an authenticated signal conveying the reference speed.

IV. Experiments

In this section we mention some of the conducted experiments whose results worth mentioning. All conducted experiments are described in our technical report [16]. While the software-in-the-loop testbed was used only for validation of safety properties, the hardware-in-the-loop testbed was used to evaluate several security-related scenarios.

A. Software-in-the-loop experiments

The SW-in-the-loop testbed can be used only for validation of properties that are not related to either hardware or real-time execution. SW-in-the-loop simulation is best used for verifying logical correctness of the executed software. In the test cases described in this section we simulated various faults and observed which errors are detected.

We automated the execution of test cases with a Matlab script that loads the list of test cases (see below), executes them and saves simulation results. For each test case type,

[7] https://rt.wiki.kernel.org/
[8] http://lxr.free-electrons.com/source/drivers/uio/uio_mf624.c

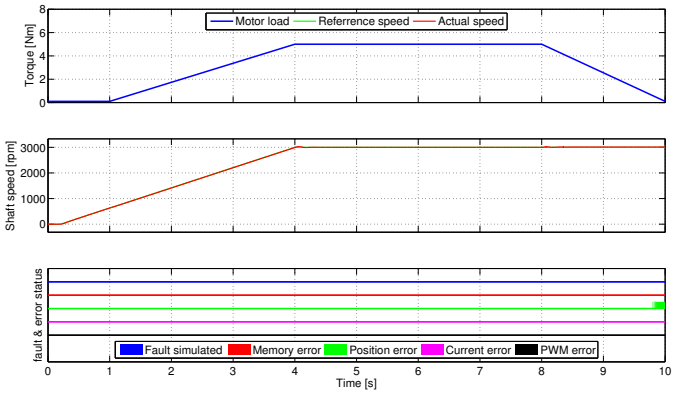


Figure 3: Test case results when no fault is simulated.

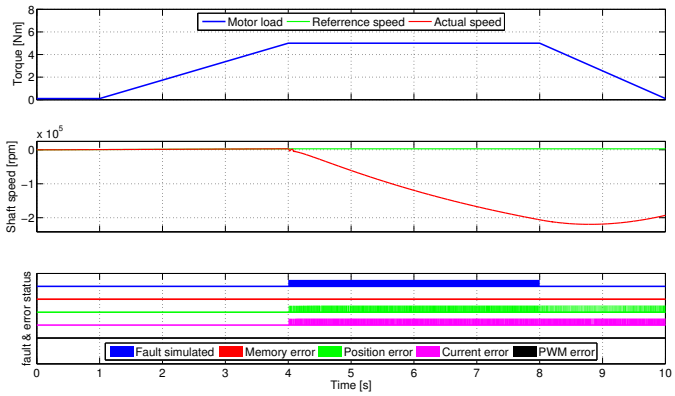


Figure 4: Position fault simulated

fault locations, types, start times, stop times and magnitude can be set. Also reference speed for the PI controller and motor load can be specified independently for each test case, but for simplicity we used the same values for all test cases. As can be seen in Figure 3, reference speed starts at 0 and increases linearly to 3000 rpm between time 0.2 and 4 s. The motor load starts at 0.1 Nm, then it linearly increases to 5 Nm in time from 1 to 4 s and then linearly decreases back to 0.1 Nm from 8 to 10 s. All simulations described below had duration of 10 s.

In the figures below, we can see the results of simulations. The bottom part of each figure shows the status of when the faults were simulated and when various errors were detected. *Thin line* means that no fault was simulated or no error was detected while *thick line* means the opposite. Colors denote different errors (see the legend in the figures).

1) No fault test case: In this test there are no faults simulated in order to test the normal operation and susceptibility to detection of false errors.

In Figure 3 we can see a false position validation error occurring near the end of the simulation. This error is signalled when there is inconsistency between simulated motor and eMotor's internal motor model. This inconsistency only appears when the motor torque decreases and it causes the measured and calculated position to slowly diverge. The false error occurs when the two positions differ more than a predefined threshold.

Because we cannot run tests with real motor, we are unable to determine which model is correct or even whether we configured the eMotor properly. This being said, we think that the current version of the eMotor prototype documentation is not clear enough with regards to the eMotor's motor model parameters (particularly definitions of used terms and vague description of filter parameters). It will be necessary to specify what are the valid operating conditions for the motor load and how to set the parameters of the eMotor's internal motor model.

2) Position sensor fault: In this case the position sensor becomes stuck at time 4 s, outputting the last value read before that event. Then it reverts back to the normal operation at 8 s.

Results in Figure 4 show, that the position sensor fault was successfully detected. The detection delay was 6.5 ms.

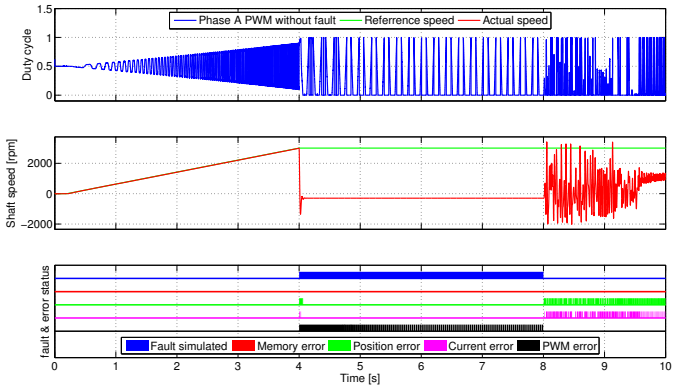


Figure 5: PWM fault simulated

It is worth mentioning that a current error was also reported (with delay of 3.5 ms) this is caused by the motor currents exceeding the range of the simulated ADC.

3) PWM wire break: In this test case the PWM signal wires for all phases break at 4 s, outputting duty cycle of 0, which results in maximum voltage applied to all phases, and then they revert back to normal operation at 8 s.

Results can be seen in Figure 5. In this figure we replaced the load profile, which was the same as in previous cases, with phase A PWM signal generated by the eMotor (i.e this signal is fed to the motor only when the fault is not simulated). The other two phase signals had similar shape so we left them out of the picture for the sake of readability. The PWM fault was successfully detected and the detection delay was one eMotor algorithm cycle (50 µs). This is the minimal possible delay, because the PWM diagnostic is implemented as a two step process alternating reading and validation in each algorithm cycle. At the beginning of the fault there are current and position errors reported, these are caused by the reaction of the motor to the loss of control and maximal voltage applied to all phases. After a short delay the motor stabilizes and those errors disappear. Because there eMotor driver does not react to safety errors yet, the implemented the controller is not stopped and as soon as the fault is over the signals affected by controller wind-up are fed to the motor and this results in erratic behavior and reported current and position errors that can be seen in the figure.

B. Hardware-in-the-loop experiments

The main goal of these experiments is to measure the execution time of the eMotor controller and see how it is influenced by the implemented safety and security functions as well as by possible faults and attacks. Note that bounded execution time is an important safety requirement for the eMotor software.

The execution time of the main controller loop (the one called by periodic ADC interrupt) is measured using on-board system timer (STM) and with each measurement the information which errors were reported during that cycle is stored. Also the fault initiation time and the time when an error was first reported are measured, allowing exact measurement of the fault detection delay.

The STM runs at 100 MHz thus having 10 ns resolution. The timer itself is 56 bit wide but only the lowest 32 bits are read and stored, because the overflow (which in the lowest 32 bits occurs every 42.9 s) can be easily handled during data post-processing. The start and end times of each loop and 4 error flags are stored in on-board RAM.

By using 1 MiB of available SRAM (unused by the original eMotor software) we can store 87 381 measurements, which at the 20 kHz control frequency results in slightly under 4.5 s of run time. After the end of the experiment, the measured data are transferred to the PC via virtual serial port provided by the TriBoard.

The tests consisted of starting the idle motor with constant 1 Nm load to 1000 rpm and holding it there for the entire measurement. In all experiments, the reference speed was set to 1000 rpm. Unless said otherwise all measurements were initiated after the motor speed stabilized.

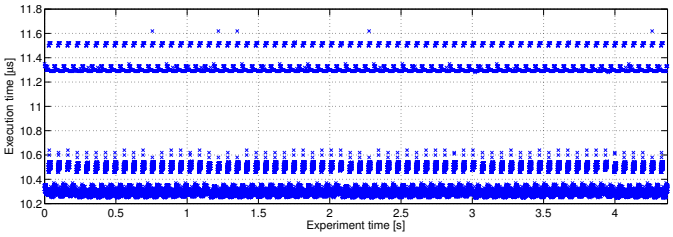
1) No safety measure enabled: In this section, we describe the experiments when no safety measure was configured in the eMotor driver, i.e. the safety measures are not compiled into the eMotor binary. Therefore, we measure the properties of the control algorithm itself. In these experiments, error information was not saved, because it is impossible for eMotor to report any errors in this configuration, resulting in increased number of measurements.

a) No fault simulated: In this case no fault was simulated. With this test, we want to investigate the properties of normal operating conditions.

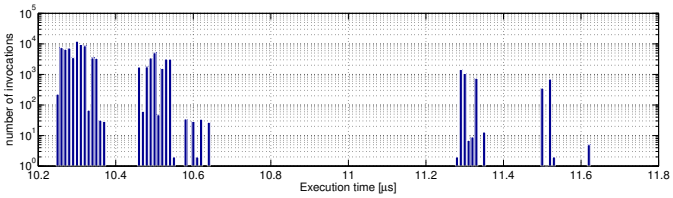
The results can be seen in Figure 6. Figure 6a shows distribution of execution time over experiment time and figure 6b shows a histogram of those execution times. Most eMotor invocations have execution time between 10.2 and 10.7 µs. Although it may not be clear from the figure it is exactly every 20th invocation that is cca 1 µs longer than the rest. This is most likely caused by the speed PI controller that runs at 1 kHz and causes cache trashing.

b) Position fault simulated: In this case the "stuck" position fault (see Section III-A3) was simulated from 0.9 s onward to check the effects of a fault on the timing properties of the eMotor driver.

Results can be seen in Figure 7. These results differ from all others, because there is a visible change in execution time



(a) Execution time



(b) Histogram of execution time

Figure 6: No safety measure, no fault

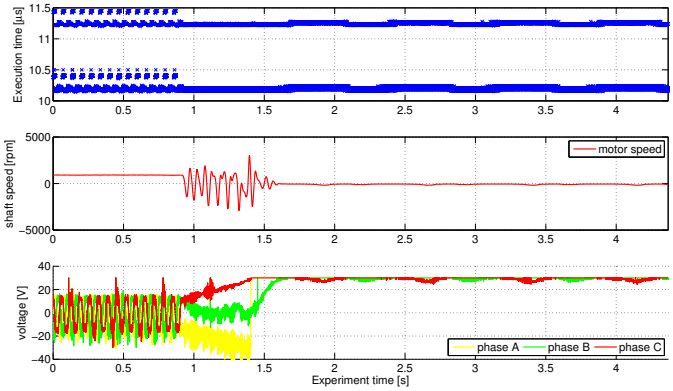


Figure 7: No safety measure position fault

(less jitter) from the moment the fault was initiated. Although this change does not violate the bounded execution time requirement, it is interesting to see how different values of mathematical operation operands result in different execution time. Moreover after short time (cca 0.7 s) all PWM channels were set to maximal duty cycle resulting in stopped motor.

2) Single safety measure enabled: In this section, we perform experiments with only one safety measure enabled (compiled in) and without faults. This allows us to see the overhead caused by the respective safety measure.

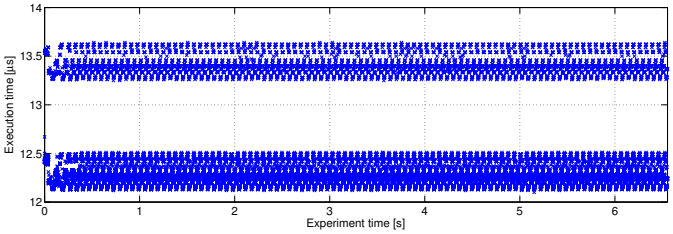
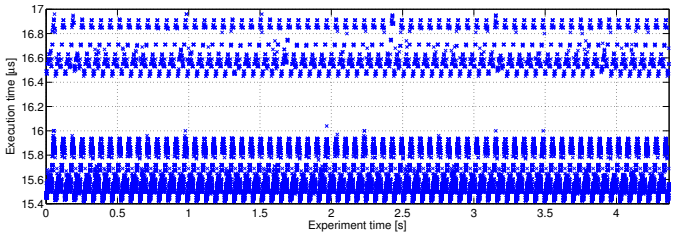


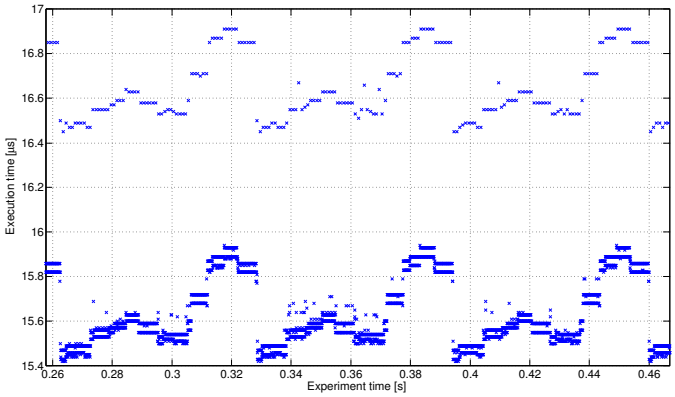
Figure 8: Position validation enabled, no fault simulated

Safety measure	Execution time [µs]
Current validation	0.1
Position validation	1.7
PWM diagnosis	0.5
Memory validation	3
All four enabled together	5

Table I: Execution time cost of individual safety measures



(a) Time series



(b) Time series (detail)

Figure 9: All safety measures, no fault

For example, Figure 8 shows the case where only the position validation was enabled and no fault was simulated. It can be seen that the position validation prolongs the execution of the control algorithm roughly by 1.5–2 µs and adds more jitter. Execution time of individual safety measures is summarized in Table I.

3) All safety measures enabled: Experiments in this section run with all safety measures enabled, as in a production system. The purpose is to see how they operate together and how do various faults influence the execution time.

First, we enabled all four safety measures and no fault was simulated. Results can be seen in Figure 9a. All safety measures combined prolong the execution of the control algorithm roughly by 5 µs and slightly prolong every other cycle due to the nature of PWM diagnostic.

Figure 9b shows a more detailed view of the same data and reveals a repeating pattern in execution time. Period of this pattern is roughly 65.7 ms. We have no explanation for why is the pattern there.

We also simulated a position fault while all four safety measures were enabled. The measured position was stuck on one value roughly at 0.9 s. As can be seen from results in

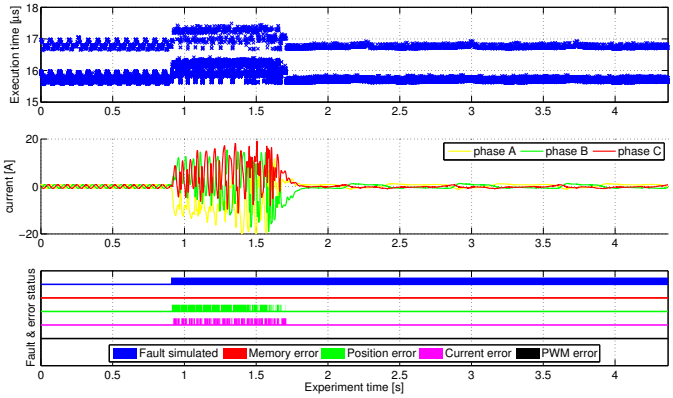


Figure 10: All safety measures enabled, position fault simulated

Figure 10, the position fault was correctly reported and the current error was reported simultaneously, which has the same cause as in the SW-in-the-loop test (see Section IV-A2).

4) CAN bus flooding: While all the previous experiments were merely safety-related, this section covers even security-related experiments. The main goal here is to determine how the additional security measures interfere with normal operation and whether they can or cannot cause a violation of a safety requirement such as bounded execution time.

In all experiments in this section the CAN bus was flooded by a continuous stream of messages with random ID, random length (0 – 8 bytes) and random data sent with the highest possible baud rate (1 Mbps) to simulate an attacker trying either to guess the key used for message authentication or to simply cause CAN bus denial-of-service. We are interested only in the effects this has on the eMotor and especially whether the extra load of the cryptographic hardware (SHE) affects eMotor's execution time. The effectiveness of the authentication scheme used in MaCAN is not of interest in this work.

First, we verified that reception of CAN messages (without authentication) based on polling the hardware rather than being notified by interrupts (typical in safety related application) does not produce a visible effects in execution time. Then, the polling method was also used in subsequent test, with message authentication which caused increased load of the SHE hardware.

In the test with message authentication enabled, the reference speed was sent over MaCAN as an authenticated signal. All safety measures were enabled and CAN bus was flooded with continuous stream of messages just like in the previous case. No artificial faults were simulated.

As can be seen from results in Figure 11 the increased load generated by checking of authenticity of those random messages has minor effect on the execution time and no safety measure reports a false error.

V. Conclusion

In this paper we presented the testbeds developed for combined validation of safety and security requirements of

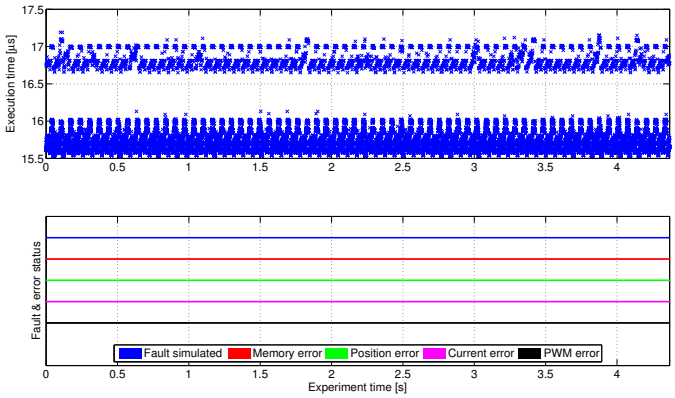


Figure 11: All safety measures enabled, no fault simulated, MaCAN control, CAN flood

an automotive application. The validation was performed by means of testing the developed item in software- and hardware-in-the-loop simulations. The tested item was a prototype of an AUTOSAR module for controlling electrical motors called eMotor, which was integrated with the message authentication protocol for the CAN bus. The hw-in-the-loop testbed was based on Linux operating system, which allowed us to achieve the needed real-time response of 50 µs and it simplified the setup of CAN bus network.

We discovered several minor shortcomings in the implemented prototypes of safety measures (or in their documentation) and reported them to Infineon.

We also integrated the eMotor software with an implementation of message authenticated protocol on CAN bus called MaCAN. Our implementation of MaCAN uses the Secure Hardware Extension (SHE) of the TriCore TC1798 CPU to accelerate cryptographic operations needed for its function. We conducted several experiments with this protocol to see the effect of envisioned attacks over CAN networks on the eMotor functionality. Our results show that such attacks have no significant influence on the eMotor functionality and that safety requirements such as bounded eMotor execution time are not violated.

Based on the experience from this case study, we argue that there is a big benefit in joining safety and security activities in the testing and validation phase of the development process. Development of the software- and hardware-in-the-loop testbeds consumes a lot of resources and since the same testbed can be used for validation of both safety- and security-related requirements, there is little benefit in having two independent teams developing the same test bed.

A full technical report containing data from all conducted experiments is available on-line [16].

Acknowledgment

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 295354 (SESAMO) and from the Ministry of Education of the Czech Republic.

This work was supported by the Grant Agency of the Czech Republic under the Project GACR P103/12/1994.

References

[1] International Organization for Standardization / Technical Committee 22 (ISO/TC 22), *ISO/DIS 26262:2010(e) – Road vehicles – Functional safety*, 2010.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces", *Proceedings of the 20th USENIX conference on Security*, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full%5C_papers/Checkoway.pdf.

[3] ISO/IEC JTC 1/SC 27 IT Security techniques, *ISO/IEC 15408:2009 – Information technology – Security techniques – Evaluation criteria for IT security*, 2008/2009.

[4] R. Isermann, J. Schaffnit, and S. Sinsel, "Hardware-in-the-loop simulation for the design and testing of engine-control systems", *Control Engineering Practice*, vol. 7, no. 5, pp. 643–653, 1999, ISSN: 0967-0661. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066198002056.

[5] O. Hartkopp and R. Schilling, "MaCAN – message authenticated CAN", in *ESCAR Conference*, Berlin, Germany, Nov. 2012.

[6] AUTOSAR, *Specification of the virtual functional bus*, R3.1 rev 5, 2010. [Online]. Available: http://www.autosar.org/download/R3.1/AUTOSAR_SWS_VFB.pdf.

[7] Infineon Technologies AG, *MC-ISAR_AUDO_UM_EmoDriver Documentation*, release V1.3, 2012.

[8] ——, *TC1798 32-bit microcontroller – Data Sheet*, V1.0, 2012.

[9] Mathworks, inc., *MATLAB Documentation*, R2012b, 2012.

[10] P. Pillay and R. Krishnan, "Modelling of permanent magnet motor drives", *IEEE transactions on industrial electronics, 537-541*, 1988.

[11] K. Belda, "Study of predictive control for permanent magnet synchronous motor drives", in *Methods and Models in Automation and Robotics (MMAR), 2012 17th International Conference on*, Aug. 2012, pp. 522–527.

[12] DCE FEE CTU, *Linux Target for Simulink Embedded Coder project*. [Online]. Available: http://lintarget.sourceforge.net/.

[13] HUMUSOFT s.r.o, *MF 624 Multifunction I/O card User's manual*, 2006. [Online]. Available: http://www2.humusoft.cz/www/datacq/manuals/mf624um.pdf.

[14] AUTOSAR, *Specification of can driver*, R3.0 rev 2, 2008. [Online]. Available: http://www.autosar.org/download/AUTOSAR_SWS_CAN_Driver.pdf.

[15] O. Hartkopp and Volkswagen, AG, "The can networking subsystem of the linux kernel", *Proceedings of the 13th iCC*, 2012.

[16] M. Kreč and M. Sojka, "SW- and HW-in-the-loop testbeds for AUTOSAR eMotor driver", Czech Technical University in Prague, Tech. Rep., 2014, Available online: https://rtime.felk.cvut.cz/publications/public/emotor-simulink-testbed.pdf.