On the Effective Use of Fault Injection for the Assessment of AUTOSAR Safety Mechanisms

Regular Paper

Thorsten Piper*, Stefan Winter, Neeraj Suri
Dependable Systems and Software Group
Technische Universität Darmstadt, Germany
{piper,sw,suri}@cs.tu-darmstadt.de

Thomas E. Fuhrman
General Motors
Warren, MI USA
thomas.e.fuhrman@gm.com

***Abstract*—The automotive safety standard ISO 26262 strongly recommends the use of fault injection (FI) for the assessment of safety mechanisms that typically span composite dependability and real-time operations. However, with the standard providing very limited guidance on the actual design, implementation and execution of FI experiments, most AUTOSAR FI approaches use standard fault models (e.g., bit flips and data type based corruptions), and focus on using simulation environments. Unfortunately, the representation of timing faults using standard fault models, and the representation of real-time properties in simulation environments are hard, rendering both inadequate for the comprehensive assessment of AUTOSAR's safety mechanisms. The actual development of ISO 26262 advocated FI is further hampered by the lack of representative software fault models and the lack of an openly accessible AUTOSAR FI framework. We address these gaps by (a) adapting the open source FI framework GRINDER [1] to AUTOSAR and (b) showing how to effectively apply it for the assessment of AUTOSAR's safety mechanisms.**

***Keywords*—AUTOSAR; fault injection; ISO 26262; robustness testing; instrumentation**

I. Introduction

In the automotive domain, many innovative functions, such as crash prevention, advanced driver assistance features, and vehicular communication systems, are enabled by software. Consequently, the software code base is growing in both complexity and size, in some cases reaching 100 million lines of code that are distributed across more than 70 electronic control units (ECUs) and interconnected by more than 5 different bus systems [2], [3]. In order to manage the complexity of these systems and to reduce the development and integration costs for new vehicle features, the automotive industry widely adopts standardized software architectures and development processes such as AUTOSAR (AUTomotive Open System ARchitecture) [4].

To warrant the trust that motorists put in the safe operation of their vehicles, many functions in automotive systems are designed and developed following stringent dependability and safety requirements. The recommended guidelines for the design, development and integration of such systems are provided by the functional safety standard for road vehicles ISO 26262 [5]. To aid automotive system developers in meeting these safety requirements, the AUTOSAR standard specifies a set of *functional safety mechanisms* [6], such as memory partitioning and timing monitoring. The verification and validation of the implementation and application of these functional safety mechanisms, which are usually supplied by third party vendors, is essential to the dependable and safe operation of these systems and to prevent hazards such as Toyota's unintended acceleration issues [7].

Among the available dependability assessment techniques, fault injection (FI) [8] is widely adopted and ISO 26262 strongly recommends its use to validate that functional and technical safety mechanisms are correctly and effectively implemented. Despite this explicit recommendation, published work on AUTOSAR FI (cf. Section III) is currently not addressing the comprehensive assessment of the correctness and effectiveness of AUTOSAR's safety mechanisms. Moreover, the predominantly used standard fault models, such as bit flips and data type dependent corruptions, are limited in their representation of timing and software faults, thus hampering their applicability to such an assessment. We argue that this situation originates from ISO 26262 recommending FI without providing appropriate guidance on the design, implementation and execution of FI experiments. The actual assessment is further hampered by the lack of an openly accessible AUTOSAR FI framework. A similar observation was made by Silva et al. [9], who conclude that "although many fault injection tools exist, none is really a ready to use tool, thus a common framework would be a major breakthrough". As state of the art, little documented experience exists on how to effectively apply FI for validating the implementation of AUTOSAR safety mechanisms.

Paper contribution

- On this background, we provide an open source, ready to use AUTOSAR FI tool[1] capable of conducting FI experiments at all layers of AUTOSAR's software architecture, i.e., application, runtime environment and basic software. Injections in source code and binary object files (i.e., white-box and black-box) are both supported.
- We report our experiences in conducting a dependability assessment of a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. The

*Author contact details: Hochschulstr. 10, 64289 Darmstadt, Germany. Phone: +49-6151-163414, Fax: +49-6151-164310

[1]The URL of the repository will be provided with the camera ready version of the paper.

assessment uncovered a real deficiency in the implementation that was subsequently acknowledged and fixed by the supplier of the safety mechanisms' implementation.
- We provide guidelines for the derivation of specific fault models, injection locations and mechanisms from the abstract AUTOSAR and ISO 26262 fault models.

Paper structure

We give a brief introduction to AUTOSAR's system model and functional safety mechanisms in Section II, followed by a review of related work on AUTOSAR FI in Section III. In Section IV, we discuss the AUTOSAR fault models that are currently used or provided by the standard. The adaptation of the open source FI framework GRINDER to AUTOSAR and the instrumentation of AUTOSAR systems is described in Section V. We demonstrate the applicability and effectiveness of FI for the assessment of AUTOSAR safety mechanisms in a case study in Section VI, including a detailed specification of the used fault models.

II. AUTOSAR: System Model and Functional Safety Mechanisms

To familiarize the reader with basic concepts of AUTOSAR, this section provides a brief introduction to its system model and functional safety mechanisms.

A. System Model

AUTOSAR systems are designed as abstract models, in which *software components (SW-Cs)* are the core building blocks. They contain functional entities called *runnables*, whose execution is triggered by recurring timers or aperiodic events (e.g., message arrival). SW-Cs interact with their environment via port interfaces that are connected in the model through a virtual functional bus (VFB).

During the system configuration phase, this abstract representation of the system is subsequently mapped to one or more electronic control units (ECUs). At an ECU, the AUTOSAR software architecture is organized in layered form as depicted in Figure 1. Closest to the hardware is the basic software (BSW) layer, which provides hardware abstractions for the microcontroller and the ECU and hosts the operating system (OS) amongst other system and communication services. The runtime environment (RTE) provides the SW-Cs of the application layer with an interface to BSW services. Moreover, the RTE implements a transparent communication abstraction that maps the virtual connections of the VFB to actual communication channels. The application layer comprises a set of SW-Cs, each of which contains one or more runnables. As runnables are not directly schedulable by the OS, they are grouped as *tasks* by the system integrator, usually taking the execution periods of the runnables and the criticality of their SW-C into account. To avoid unintentional interactions between tasks of different criticality (e.g., by error propagation), AUTOSAR offers a set of functional safety mechanisms to monitor and isolate tasks, which are described in the following subsection.

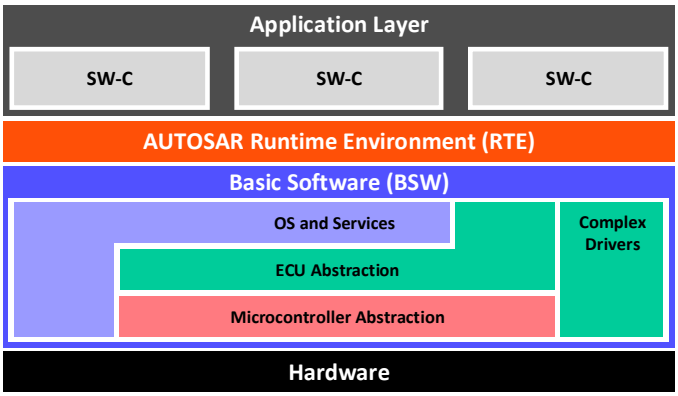


Fig. 1. The AUTOSAR software architecture.

B. Functional Safety Mechanisms

For the co-existence of tasks with different criticality, i.e., different automotive safety integrity levels (ASILs), on the same system, ISO 26262 requires *freedom from interference* in both space and time. This means that lower ASIL tasks must not interfere with higher ASIL ones, for example through error propagation. To realize freedom from interference, isolation mechanisms are used to establish fault containment regions. The *Overview of Functional Safety Measures in AUTOSAR* [6] (a document which recently superseded the *Technical Safety Concept Status Report* [10]) specifies the following functional safety mechanisms to assist with the prevention, detection and mitigation of hardware and software faults to ensure freedom from interference between tasks.

Memory Partitioning. To prevent low ASIL tasks from wrongfully accessing memory of higher ASIL tasks (e.g., by corrupting their content), arbitrary tasks may be grouped to so-called *OS applications* which are subsequently executed in separate memory partitions, i.e., the code executing in one partition cannot modify memory of a different partition. Memory partitioning allows to protect read-only memory segments as well as memory-mapped hardware.

Timing Monitoring. The safety of a system often depends on the timely execution of actions and reactions such as object recognition for crash avoidance or crash detection for airbag inflation during an accident. The OS provides a set of monitoring mechanisms to detect conformance (e.g., whether tasks are dispatched at the specified time) or deviance (e.g., when tasks violate their execution time budgets or monopolize OS resources).

Logical Supervision. To detect control flow errors, i.e., any divergence of a program's execution sequence from its error-free execution sequence, checkpoints are placed throughout a *supervised entity* at design time. When encountering a checkpoint, the Watchdog Manager is notified and verifies that the sequence of encountered checkpoints is valid. In addition, temporal monitoring mechanisms such as aliveness and deadline monitoring are implemented using checkpoints.

End-2-End Protection. To ensure the integrity of data transmitted between SW-Cs, both within the same ECU as well as across networks, the end-2-end protection library enables the sender to protect data prior to transmission and the receiver to

detect and handle errors in the communication link at runtime. Several standardized profiles offer different sets of protection mechanisms (e.g., CRC, sequence counter, alive counter) that are suitable for diverse requirements.

The functional safety mechanisms (except for end-2-end protection) are implemented by the OS or as BSW services, and aim at detecting and mitigating the erroneous behavior of tasks. Given the broad scope of these mechanisms ranging from the BSW to the application layer, any framework for their comprehensive FI-based assessment naturally benefits from access to the complete AUTOSAR stack for the flexible placement of fault injectors and monitors of system behavior.

III. Related Work

Following a standalone exposition of state of the art AUTOSAR FI techniques, we summarize its viability for assessing AUTOSAR safety mechanisms and comment on the utilized fault models.

Simulation-based FI

In 2010, Lanigan et al. [11] used the commercial off-the-shelf (COTS) tool Vector CANoe to build a FI framework for AUTOSAR. This early work on AUTOSAR FI primarily comments on the feasibility and practical obstacles of building such a framework and does not report any specific results of dependability analyses of AUTOSAR-based (sub-)systems or provide specific fault models. In CANoe, AUTOSAR systems are executed in a simulation environment on a host PC. Faults can be injected in various components of the basic software (BSW) layer via a set of hooks that either suppress calls to certain API functions (suppression hooks) or manipulate specific data structures of an API (manipulation hooks). Hooks are manually placed in the code and the actual implementation of the fault models is provided by an external library that is linked to the simulation environment. The authors conclude that "CANoe is a suitable fault-injection environment for some faults, but that other faults cannot be represented using the level of abstraction that CANoe provides".

Another simulation based FI framework is from Baumgarten et al. [12] where application-level software components (SW-Cs) of existing AUTOSAR models are annotated and extended by so called *fault ports*. These extended models are translated to C-code using dSpace TargetLink which is a code generation tool that operates on Simulink and Stateflow models. Faults are implemented as additional code in the TargetLink model and triggered during simulation by signaling the corresponding fault port(s). Consequently, faulty behavior is added in form of functional blocks, which are then able to disturb the normal behavior according to the provided fault implementations. Annotating models with fault ports is currently a manual process and the authors leave automated annotation for future work. The actual simulation can be performed on two different levels, either in the integrated simulation environment of dSpace SystemDesk to simulate the whole AUTOSAR architecture or at the software unit level using test tools for C code. The authors evaluate their approach in a front-lights controller setting, considering physical chip damage, stuck-at, crash and message loss as fault models. The effects of the faults are emulated on the application layer by disabling the execution of SW-Cs, by forcing port interfaces to specific values, or by omitting messages.

To identify threats to functional safety early in the system design stage, Pintard et al. [13], [14] propose fault injection analyses (FIA) on pre-implementation design artifacts to complement fault injection experimentation (FIE) on an actual (prototype) implementation. To conduct FIA, a functional model is created from the system requirements using data flow diagrams, state charts, UML or AADL, depending on the available level of detail that the specification provides. Using this model, failure modes of functions or components are assumed and their effect on the system operation analyzed, similar to failure mode and effect analysis (FMEA) and failure mode, effect and criticality analysis (FMECA). Subsequently, the results of FIA are used as guidance to perform FIEs, for example by identifying critical components.

Vedder et al. [15] develop a method and tool for combining property-based testing (PBT) and FI for testing safety-critical systems. PBT automatically generates test cases from a specified property of a system, i.e., it generates test input values and at the same time acts as an oracle for the expected output. The authors use the commercially available tool QuickCheck for generating the test cases in conjunction with their own tool FaultCheck, a C++ library with an optional wrapper for C, that conducts the actual injections. The tool is evaluated on an AUTOSAR E2E library implementation in an isolated simulation environment.

Overall, while simulation-based approaches are useful to conduct FI analyses and experimentation in the early stages of system development, they are inherently limited in their representation of detailed lower-level fault models and timing conditions necessary for addressing real-time operations. As the meaningful application of many safety mechanisms depends on an exact representation of time and real-time constraints, a comprehensive assessment of timing-dependent safety mechanisms is infeasible using simulation-based approaches.

Hardware-based FI

Cunha and colleagues [16], [17] developed the fault injection tool csXception® for scan-chain implemented fault injection (SCIFI) on an ARM Cortex-M3 microcontroller, which is capable of injecting faults at the hardware level (i.e., processor and auxiliary registers, flash memory and SRAM). The considered fault models are bit flips, reset values and specific values, which can be activated based on various fault trigger conditions. The tool is evaluated in an anti-lock braking system (ABS) case study where the focus of the evaluation is the general application of the tool rather than addressing actual dependability properties of the ABS system.

Salkham et al. [18], [19] present a FI framework implemented as AUTOSAR software components (SW-Cs) and complex device driver (CDD). The FI controller and monitoring services are implemented on the application layer as SW-Cs and isolated from the rest of the system using memory partitioning. The CDD contains a collection of FI modules that implement specific error types for each target component. The approach is evaluated in two example scenarios: communication errors that are modeled by disabling the CAN bus circuit

and NVRAM errors that are modeled by corrupting CRC bits. While the experiment control logic is implemented as software elements, hardware mechanisms are utilized to perform actual injections.

Hardware-based FI commonly has the advantage of handling low level implementation details. However, to cover dependability properties that are related to the interaction of hardware and software elements or to accelerate experiments [20], [21], e.g., by using software states as trigger conditions for injections, software based control as in [18], [19] is often required. To exercise precise control over software interactions and software timing in our study, we also chose to implement the injection mechanisms in software.

Software-based FI

Islam et al. [22] proposed the BeSafe framework for benchmarking the functional safety of AUTOSAR systems on three different abstraction layers: model, software and hardware. The benchmark framework supports FI at the software level through a proprietary tool called B-FEAT. The tool is capable of intercepting calls to/from SW-C interfaces, facilitating data type and fuzzing error models to evaluate the robustness of SW-Cs. The resilience of SW-Cs with respect to transient bit flip hardware faults is benchmarked using the FI tool GOOFI-2 [23]. On the model level the tool MODIFI [24] is used to conduct dependability evaluations of Simulink models early in the development phase, mainly to assess error detection and recovery mechanisms. The considered fault models are data type and fuzzing on the software level, bit-flips on the hardware level and bit-flips and sensor faults on the model level. Unfortunately, the preliminary evaluation of the framework has not been conducted on an AUTOSAR system. However, the reported assessment of a CRC mechanism suggests that the framework potentially could be used to assess AUTOSAR's end-to-end (E2E) protection mechanisms.

With the aim to assess the robustness of AUTOSAR COTS components that are available as binaries only, Islam et al. [25] present a technique and tool prototype for binary-level fault injection (BLFI). Contrary to the binary-level instrumentation approach presented in [26], no AUTOSAR specific information is used to drive the instrumentation, thereby expanding its applicability to all AUTOSAR layers including the BSW. The broader application scope comes at the cost of losing the reference to the underlying system model, which impacts the usability of the instrumentation if the system model is used to select instrumentation locations. The tool is evaluated on a blinking LED warning system, using data-type based and fuzzing fault models.

Summary Comments

We observe that the fault models used in related work are predominantly *standard* fault models[2], such as bit flips and data type dependent corruptions, that have been adopted from studies for different target systems and application scenarios. It is surprising that no *AUTOSAR-specific* fault models are used, as the choice of fault models heavily impacts the effectiveness of the FI experiments [27], [28] and domain-specific fault models are expected to yield better results.

We also observe that, despite an explicit recommendation in the ISO 26262 standard, no studies on FI-based assessments of the mechanisms in AUTOSAR's technical safety concept exist, apart from the E2E library.

Finally, we observe that many FI approaches are simulation based and as such unable to adequately represent real-time properties and timing behavior of the system. As we demonstrate in Section VI, assessing real-time properties is essential to many safety-critical applications and AUTOSAR's timing monitoring mechanisms, which constitute a significant fraction of the technical safety concept.

IV. AUTOSAR Fault Models

A fault model is a representation of a possible internal or external fault [29] that a system may be exposed to. In the context of FI experiments, a fault model is characterized by the fault location (*where* to inject), the fault type (*what* to inject) and the fault timing (*when* to inject) that possibly also includes an expected workload or system state. The selection of *representative* fault model(s), i.e., faults that may and do occur during the development and operation of the tested system, is crucial as non-representative faults can significantly affect the injection results [28] by hampering (a) their accuracy and usefulness [30] and (b) the effectiveness of the experiments to reveal robustness vulnerabilities [27]. Consequently, developing and using a realistic fault model is one of the biggest challenges for any FI schema [9].

The use of standard fault models such as *bit flips* and *data type dependent corruptions* is currently predominant in related work on AUTOSAR FI (cf. Section III). While these established fault models adequately represent specific abstraction levels and classes of faults (and their effects), their applicability to represent complex software and also timing faults is limited. For the *effective* assessment of AUTOSAR's safety mechanisms, the use of standard fault models should therefore be combined with fault models that allow a more direct mapping of application level software and timing faults.

Before the release of the *Overview of Functional Safety Measures in AUTOSAR* [6] end of 2014, no central document provided documentation on the applicable fault models for the evaluation of AUTOSAR's functional safety mechanisms. Moreover, representative fault models are invariably based on deep domain knowledge such as known failures, identified hazards and (non-functional) safety and dependability requirements–in summary information that is not necessarily publicly available to the research community. We argue that both of these factors have contributed to the slow adoption of more representative software and timing fault models for AUTOSAR FI. In addition, an opinion that has been prevalent for a long time in the automotive community is that software faults are generally covered by and detected during the verification phase of development, due to the systematic nature of these faults. Given the growing complexity of automotive software, it is questionable whether this view still holds, or for how long.

Even though the release of applicable fault models for AUTOSAR's functional safety mechanisms in [6] is a step in

[2]In Section IV we further elaborate why the "standard fault models" are still prevalent.

the right direction, the models are still at a very abstract level as they are directly adapted from ISO 26262, which is a generic standard for road vehicles and not AUTOSAR in particular. Concrete examples of software defects are thus omitted and instead only the effects of faults are exemplified. To give an example, the only information that AUTOSAR and ISO 26262 provide for fault models applicable to *timing monitoring* is "blocking of execution, deadlocks, livelocks, incorrect allocation of execution time, and incorrect synchronization between software elements". This information serves as suggestion for *what* to inject at best, while guidance on the actual application of these fault models, i.e., *where* and *when* to inject, is still missing.

Other documents, such as the *Description of the AUTOSAR standard errors* [31], improve on this but are still work in progress. While having the purpose of giving an overview of dysfunctional behavior, clarifying error handling mechanisms and giving the failure modes coverage of the different mechanisms, the scope of the document is currently limited to the CAN communication stack and the memory stack. As the specification of AUTOSAR fault models is still an ongoing process, one may fall back to studies on representative software faults from other domains in the meantime [21], [28], [32].

Surprisingly, the consideration of simultaneous fault models is currently missing completely in AUTOSAR, although ISO 26262 explicitly considers dual-point and multi-point failures, i.e., failures resulting from the combination of two or several independent faults that leads directly to the violation of a safety goal [5]. Until the standard incorporates multi-point faults, we refer to the work of Winter et al. [33], who provide a comprehensive study and new approaches to assess the degree to which systems are vulnerable to multiple-fault conditions.

As consequence of the ongoing development of the AUTOSAR standard and the reliance on deep domain knowledge to specify fault models, any FI framework utilized for the assessment of the functional safety mechanisms should be easily and flexibly extensible to account for future standard revisions and domain specific requirements. For the FI framework that we present in Section V, we therefore use a fault model library to offer this flexibility, while at the same time enabling the reuse of fault models that have already been implemented.

V. Applying the Open Source FI Framework GRINDER for AUTOSAR FI

Following the discussions on AUTOSAR-specific FI and fault models, we now detail the application of the open-source FI framework GRINDER for AUTOSAR FI. After presenting the FI workflow, we discuss GRINDER's adaptation to our AUTOSAR evaluation environment and comment on the instrumentation methodology for using GRINDER with AUTOSAR.

The workflow of an FI experiment typically comprises the following three phases.

1) **Configuration:** The workload is prepared and the target system is instrumented with injector(s) and detector(s) according to the experiment's specification. The workload should match the evaluation target and also ensure that all injection(s) are activated by meeting their respective trigger condition(s) (i.e., *when* to inject).
2) **Execution:** The target system is executed until the injection of fault(s) and the collection of perturbation data is successfully completed, or until a stop criterion (e.g., a timeout) is satisfied.
3) **Evaluation:** The experiment outcome, for example logs and traces that were collected during its execution, is analyzed.

As this workflow offers much potential for automation, FI frameworks are typically employed for the automated and efficient execution of campaigns (series of experiments) with the positive side-effect of precluding human error from adversely affecting experiment results.

For the assessment of AUTOSAR's functional safety mechanisms, we faced the challenge of which framework to use. Specifically for AUTOSAR FI, none of the existing frameworks is publicly available, nor do the existing frameworks fulfill the requirements for such an assessment (cf. Section III). While considering publicly available generalist tools such as FAIL* [34] and LLFI [35] as potential alternatives, we realized that the adaptation of these tools was infeasible. FAIL* targets architecture simulators for x86 and ARM that do not suit the PowerPC architecture of our evaluation system, and LLFI relies on the LLVM compiler infrastructure, whose use would require the modification of large parts of the existing AUTOSAR development environment and tool chain. In conclusion, the adjustment of openly available tools to suit the purpose of our assessment would likely have outweighed any effort for re-implementing a custom tool from scratch. Consequently, we adapted the open source FI framework GRINDER to AUTOSAR, which entailed modest implementation and configuration overhead [1]. In the following two subsections, we detail the adaptation of GRINDER to an AUTOSAR system and the instrumentation of AUTOSAR systems for FI experiments.

A. Adapting GRINDER to an AUTOSAR System

GRINDER is an open source general-target FI tool that is written in Java and built around an extensible architecture with a simple interface for target abstraction, extension and customization, and which has reusability as one of its primary design goals. With the intent of making a ready-to-use AUTOSAR FI framework publicly and freely available, we will release our AUTOSAR adaptation of GRINDER as open source[3].

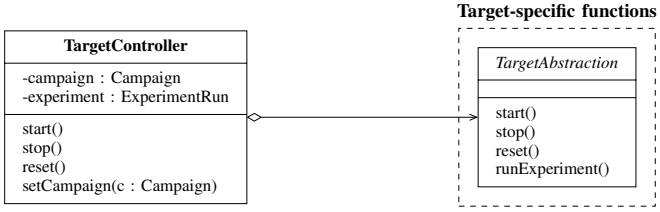


Fig. 2. The *TargetController* class and the *TargetAbstraction* interface [1].

GRINDER is adapted to a new target system (e.g., an AUTOSAR system) by providing a target-specific implementation of the so-called *TargetAbstraction* interface, by which

[3]The URL to the public repository will be provided with the camera ready version of the paper.

GRINDER interacts with target systems (cf. Figure 2). The TargetAbstraction specifies a simple set of target-specific functions that GRINDER's *TargetController* class requires to control target systems: *start()*, *runExperiment()*, *reset()* and *stop()*. The specification was driven by the observation that on an abstract level the progression of FI experiments across different tools and targets is the same: target initialization, workload invocation, fault injection and data collection [1], closely matching the description of the FI workflow in the beginning of the section. After *starting* the target, an experiment is *run* and data is collected for analysis. After experiment completion, the target is *reset* to a known stable state to avoid the impact of undetected residual injection effects on subsequent experiments. These steps are repeated until each experiment of the current campaign has been executed. Afterwards, the target is *stopped* and exchanged or reconfigured, if this is required for subsequent experiments.

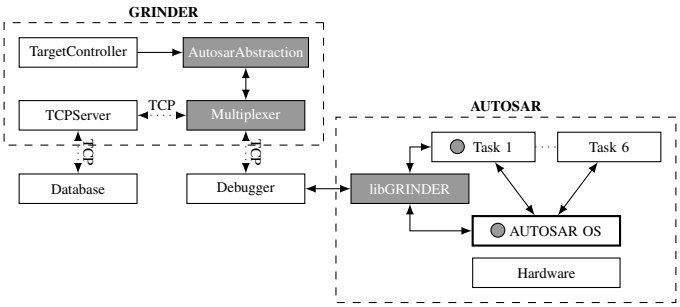


Fig. 3. Adapting the GRINDER FI framework to AUTOSAR [1].

GRINDER's architecture resembles the general FI tool architecture presented by Hsueh et al. [8]. Its integration in an AUTOSAR FI setting is depicted in Figure 3 where components that had to be either developed or adapted for GRINDER's use with the AUTOSAR system, i.e., the *AutosarAbstraction*, the *Multiplexer* for communication with the target, and the fault model library *libGRINDER*, are colored gray. The details on each of these components are provided below.

Using GRINDER, the experiments are directly configurable regarding the location of fault injectors, for the placement of monitors, for selecting employed fault models and for logging of campaign data. The experiment and campaign configurations are stored in a MySQL-compatible *Database* (e.g., MariaDB). To transmit the experiment configurations from GRINDER to the target and experiment data back from the target to GRINDER, a communication channel between GRINDER and the target is used. The *TCPServer* provides a TCP-based communication interface to (a) handle incoming configuration requests from the target by fetching and sending the configuration of the next executable experiment from the database and (b) store log data from the target for the currently executing experiment in the database.

The AUTOSAR target system that GRINDER is adapted for runs on a Freescale XKT564L evaluation board[4], which hosts a 32-bit dual core Power Architecture microcontroller. As the XKT564L target is not equipped with an Ethernet interface to directly interact with GRINDER's TCPServer, the board is connected to a host computer via its JTAG/Nexus hardware debugging interface. On the host computer, the Green Hills MULTI[5] *Debugger* is utilized by GRINDER to interact with the hardware, and a *Multiplexer* handles interactions of the AutosarAbstraction and the TCPServer with the target through the debugging interface and vice versa. On the AUTOSAR target, the generic and extensible C library *libGRINDER* implements a compatible communication interface for debugger-based message exchange. The library further provides preconfigured injector, detector, and logging logic for the use in interceptors, i.e., probes in the target system that can be used to inject faults or monitor the system's state (they are indicated by gray circles in Figure 3).

The AutosarAbstraction implements GRINDER's TargetAbstraction interface as follows.

- **start()** initializes the experiment environment by starting a new instance of the MULTI debugger, connecting to the debugger via TCP and establishing a connection between the debugger and the evaluation board using the debugger's *connect* command. A valid target configuration in MULTI is required for the connect command to succeed.
- **runExperiment()** prepares the target system by verifying that the correct binary is loaded and starts the execution of the target system. Furthermore, a *variable watch*[6] is used by the target to indicate a communication request, for example to retrieve configuration options or to store log information. As runExperiment() has full access to the debugger's features, additional functionality may be implemented if needed.
- **reset()** instructs the debugger to halt the system and set it to the initial state. Since the debugger is always able to reset the target system, this method works well to reset the entire system without further interaction.
- **stop()** terminates the experiment environment by disconnecting the target system and shutting down the MULTI debugger.

It is noteworthy, that the presented AutosarAbstraction is applicable for debugger-based interaction of the host computer and the AUTOSAR system, using Greenhills MULTI and the Nexus debug interface, as a direct TCP connection is unavailable. While both, the MULTI toolchain and the Nexus interface, are widely used, other AUTOSAR evaluation setups may feature slightly different tools or interfaces. For such scenarios, we estimate that the adaptation of the provided AutosarAbstraction to a different setup entails low overhead as only the communication channel needs to be adapted accordingly.

B. Instrumenting AUTOSAR Systems for FI: What is special about AUTOSAR?

Fault injection relies on mechanisms to access and modify the actual data- and control-flow within a system. These mechanisms are typically implemented as *interceptors* that are inserted in the system by instrumentation. Interceptors may implement arbitrary functionality, such as altering data (e.g.,

[4]http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=XKT564L

[5]http://www.ghs.com/products/MULTI_IDE.html

[6]A hardware breakpoint that constantly inspects a watched variable during program execution for write access.

bit-flips, fuzzing, etc.), modifying the control flow (e.g., call other functions) or monitoring the system (e.g., for logging data).

The instrumentation of AUTOSAR systems is complex due to the following factors [26]:

- AUTOSAR systems are developed as models that highly abstract from the actual implementation, which comprises a mixture of tool-generated and hand-written code. As consequence, instrumentation at the model level cannot leverage implementation specifics (thereby limiting customizability), while instrumenting the implementation, i.e., mostly tool-generated code, is a tedious process if performed manually (thereby limiting usability).
- Given the degree of abstraction, elements of the model often have no singular representation in the implementation. Thus, the view on the system that the model provides is inherently different from the actual implementation. Moreover, as AUTOSAR promotes the integration of white-box and black-box components by various suppliers, who distribute their software either as source code or binary-only (to protect their intellectual property), the chosen instrumentation approach should work on source code and binary objects.

For the assessment of the functional safety mechanisms, the instrumentation of all AUTOSAR layers is beneficial, as it enables the injection of faults that manifest at the SW-C, RTE or BSW layers, and the flexible placement of monitors to observe system behavior and fault effects. As shown in [26], an automated framework, which leverages development artifacts from the system model and the implementation to drive the instrumentation process, can help with the instrumentation on the SW-C and RTE levels.

On the BSW level and specifically for the OS, a different approach has to be chosen, as no standardized model of the OS exists to augment and drive the instrumentation process. Our analysis of actual AUTOSAR OS code has shown that the situation is further complicated by the extensive use of macros throughout the OS for performance reasons. This directly impacts the applicability of static analysis to derive potential instrumentation locations, e.g., for interface injections [36], as constructs that syntactically look like function calls can turn out to be macros that lack type safety and have no type declarations. Macros, in contrast to actual functions, do not have explicit signatures (prototypes), return types or parameter types. Despite these impediments, a tool-supported instrumentation still has the advantage of the automated generation of interceptor code. Hence, we have chosen to extend the static analysis tool CIL (C Intermediate Language) [37] with a plug-in for the instrumentation of AUTOSAR OS source code, which we complement with manual instrumentation when needed. For binary level instrumentation, the approaches presented in [25], [38] are conceivable.

We do emphasize that recurrent instrumentation can negatively impact the experiment efficiency especially when running experiments on actual embedded hardware due to the necessary time-consuming re-flash cycles. Our approach to avoid re-flashing, or at least to minimize its usage, is to instrument the system upfront with all interceptors that are potentially required for various test campaigns and to selectively enable or disable them at runtime in a pre-experiment configuration phase. This is achieved by assigning a unique identifier to each interceptor, whose state (on or off) can be configured on a per-experiment basis.

VI. Fault Injection Case Study

Having outlined the FI framework and instrumentation approach, we now present our case study where we demonstrate the assessment of AUTOSAR's timing monitoring safety mechanisms in two scenarios using GRINDER. As mentioned earlier, the intent is to highlight the viability of the approach and its effectiveness in locating a bug in the timing monitoring implementation of a commercial AUTOSAR OS. For simplicity of illustration and communicating the insights, we focus on a small subset of the conducted FI experiments.

In the conducted experiments, we concentrate on assessing safety mechanisms specifically related to timing monitoring. This choice was motivated by the following factors.

- *Timing monitoring* is one of the newest safety mechanisms added in AUTOSAR and, except for a programmable timer interrupt, the monitoring functionality is entirely software based. With the discussion (cf. Section III) on the limitation of classical FI to address timing issues, this constitutes a good target to detail how faults related to task timing can be accurately injected with software.
- *Memory protection* is based on COTS hardware that is widely applied in other domains. Except for project-specific misconfigurations, we did not see much potential for software-related dependability issues.
- *Logical supervision features*, such as aliveness and deadline monitoring, that are implemented by a watchdog are well-established and were already used in systems based on the AUTOSAR predecessor OSEK.
- *End-2-end protection* is library-based and does not necessarily require an AUTOSAR environment for testing, as was shown by Vedder et al. [15].

In summary, we considered the assessment of timing monitoring to offer a pertinent case study for FI applicability and of its effectiveness.

AUTOSAR's timing monitoring consists of four discrete mechanisms that, in general, can be used independently, but may require a combined use to attain guarantees on specific system timing characteristics.

- *Execution time monitoring* checks a task's execution time against a fixed time budget. When the budget is exceeded without the task finishing its execution, a timing error is detected.
- *Inter-arrival time monitoring* monitors a task's activation frequency within a statically configured time frame. When an activation threshold is exceeded, a timing error is detected.
- *Resource locking time monitoring* checks the locking time of resources by tasks against a fixed time budget. When the budget is exceeded without the task releasing the resource, a timing error is detected.
- *Interrupt locking time monitoring* checks the locking time of interrupts by a task against a fixed time budget.

When the budget is exceeded without the task re-enabling interrupts, a timing error is detected.

For the case study, targeting correctness and robustness as the drivers, we illustrate the FI approach and its evaluation over two example scenarios (i.e., a simple case of execution time monitoring and a complex timing interaction scenario) to demonstrate its broad applicability.

Scenario 1: Task timing errors are provoked to (a) assess the *correctness* of the error detection and error mitigation of *execution time monitoring*, i.e., whether the mechanism detects the timing errors and mitigates their effect, and (b) analyze *error propagation* within the system with and without timing monitoring enabled.

Scenario 2: The interaction between *execution time monitoring* and *resource locking time monitoring* is investigated. Both mechanisms share a common timer (embedded hardware usually has a limited number of timers), and the timer for execution time monitoring is reset and overwritten when a monitored resource is acquired. Arbitrary task timing and resource usage patterns are injected to (a) assess the *correctness* of the mechanisms, and (b) to assess their *robustness* by aiming to provoke situations in which the re-activation of the original timer fails.

The following sections detail the progression of the FI setup and its execution. We start with the detailed specification of the used fault models. Subsequently, the evaluation setup is presented in Section VI-B and the results of the evaluation are discussed in Section VI-C.

A. Deriving Fault Models for the Case Study

In the following, we describe how we derived the fault models used for the case study. As the fault models provided by AUTOSAR and ISO 26262 are very abstract, the intent is to provide these examples as guideline for other FI-based assessments of AUTOSAR safety mechanisms.

Scenario 1 - Assessing Execution Time Monitoring

In the first scenario, we aim to trigger timing errors of arbitrary tasks by altering their control flow to either call a timed loop, thereby extending their runtime by a fixed offset (transient fault), or an infinite loop, thereby blocking execution completely (permanent fault). The trigger condition of the injection should be freely configurable in order to analyze the effects of error propagation throughout the system at workload-dependent times. The fault model for this scenario is specified as follows.

- *Fault type:* A loop that consumes a defined (possibly infinite) amount of CPU time to inject transient and permanent timing faults with high accuracy.
- *Fault location:* In the control flow of a monitored task, e.g., in its implementation (when source code is available) or its invocation.
- *Fault timing:* During different phases of the workload. The injection is triggered when a counter of the number of task invocations reaches a configurable threshold.

Scenario 2: Assessing Timing Monitor Interactions

For the second scenario, we manually reviewed the source code of the implementation of timing monitoring in a commercial AUTOSAR OS to identify potential robustness issues that we could further analyze with FI. We should highlight, that although having profound knowledge of AUTOSAR and the C programming language, this was our first encounter with AUTOSAR OS code. As such, our analysis was not influenced or guided by in-depth knowledge, and we chose the generic approach of identifying assumptions that were potentially made by the developers (OS experts could have used a more refined approach). We checked for assumptions regarding the outcome or return value of an operation, shared resource usage, potential time-of-check to time-of-use issues (i.e., race conditions), assumptions on variable initialization or state, and more, which resulted in eleven potential FI targets. In the following, we detail the FI target that we use in our case study and that uncovered a deficiency in the implementation of the timing monitoring safety mechanisms, which was subsequently acknowledged and fixed by the supplier.

In our code review, we had noticed that resource lock monitoring uses the same timer as execution time monitoring to detect resource lock errors accountable to the excess of a lock time budget. When a task acquires a monitored resource, the timer for execution time monitoring is reset and the remaining execution time budget is stored and compared to the lock budget of the resource. The smaller value (i.e., the shorter time frame) is then used as the new timeout value and the resource lock timer is activated. Whenever the resource lock is released by the task, execution time monitoring is re-activated. Although sharing a timer, or resources in general, is common in embedded systems, it also increases complexity due to synchronization issues.

To assess whether any resource usage and task timing patterns could potentially lead to a failure of the re-activation of the execution time monitor, we iterated over various timings for the resource lock time, execution time and the resource lock time budget. To emulate different time budget configuration efficiently, we directly inject in the monitoring mechanism's kernel data structure. This FI-based approach has two advantages over a conventional approach. Firstly, it enables the injection of *unexpected* budget configurations for the purpose of robustness testing, which is restricted by the configuration tool to sound budget settings. Secondly, it greatly accelerates the assessment, as the workload (i.e., task timing behavior) and the configuration of timing monitoring budgets can be directly adjusted between experiments. Normally, changing the configuration of timing monitoring on the target is a tedious and time-intensive process. It entails modifying the configuration in a GUI, generating kernel source code, compiling the binary image and flashing it to the target. The same process applies to modifications of a task's timing behavior with additional adjustments needed for the actual implementation.

For the assessment of timing monitor interactions, we employ two fault models specified as follows. The first fault model is adapted from the previous scenario to flexibly induce different resource lock times during a task's execution. This is achieved by altering its control flow between the acquisition and release of a monitored resource.

- *Fault type:* A loop that consumes a defined (possibly infinite) amount of CPU time to emulate transient and permanent timing faults with high accuracy.
- *Fault location:* Between the acquisition and release of a monitored resource.
- *Fault timing:* Between the acquisition and release of a monitored resource. The injection is only triggered on the first acquisition of a resource.

The second fault model aims to modify the configured resource lock budget by injecting arbitrary budget values in the monitoring mechanism's kernel data structure as follows.

- *Fault type:* An arbitrary, potentially unsound resource lock budget. A series of lock budgets is generated by iterating over a fixed time interval with a configurable step size. Of this series, one budget is injected per experiment.
- *Fault location:* In the kernel data structure holding the monitored task's timing characteristics and budget configuration.
- *Fault timing:* Before or upon the resource allocation of the monitored task. The injection must have occurred before the budget configuration is evaluated by the monitoring mechanism.

B. Evaluation Setup

The case study example is a simple adaptive cruise control (ACC) system and depicted in Figure 4. The ACC comprises four software components that contain one or two runnables with periods of 10 ms and 40 ms each. Further, the SW-C *Environment* provides environmental stimuli to the ACC. It is noteworthy that although this case study example is purely hypothetical and does not represent any particular real design it is intended to represent plausible mixed-IP[7] and mixed-criticality integration scenarios relevant to the automobile industry.

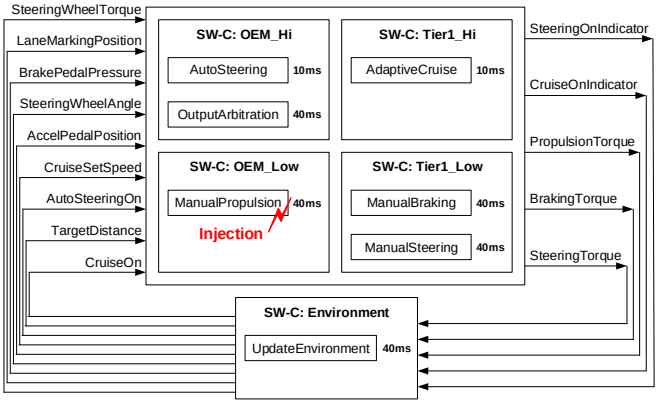


Fig. 4. The adaptive cruise control (ACC) case study example.

Table I lists the runnable to task assignment and the configuration of the six tasks of the system. The tasks are ordered from highest to lowest priority, which AUTOSAR schedules following a fixed-priority preemptive approach, i.e., when a tasks becomes ready that has a higher priority than the currently running task, the running task is preempted and the new task is executed.

[7]Mixed-IP systems integrate intellectual property (IP) by various suppliers.

TABLE I. TASK CONFIGURATION OF ACC CASE STUDY EXAMPLE.

Task name	Priority	Runnable(s)
OEM_Hi_10ms	100	AutoSteering
Tier1_Hi_10ms	90	AdaptiveCruise
OEM_Hi_40ms	80	OutputArbitration
Environment_40ms	70	UpdateEnvironment
OEM_Low_40ms	60	ManualPropulsion
Tier1_Low_40ms	50	ManualBraking, ManualSteering

In order to enable error reporting and a flexible reaction to different errors, AUTOSAR specifies the so-called *ProtectionHook* interface as part of its error handling process [39], [40]. The ProtectionHook is invoked whenever one of the safety mechanisms detects an error. The detected *error type* is passed as parameter, based on which further analysis and mitigation steps may be initiated. The user-supplied return value of the ProtectionHook defines whether the OS performs further actions (e.g., task termination or ECU shutdown) or ignores the error. Throughout the case study, we use the information provided by the ProtectionHook to determine if and when an error was detected. Moreover, to flexibly enable or disable timing monitoring (e.g., to compare system behavior or error propagation), we modify the return value of the ProtectionHook depending on the experiment configuration using an injector.

C. Experimentation and Results

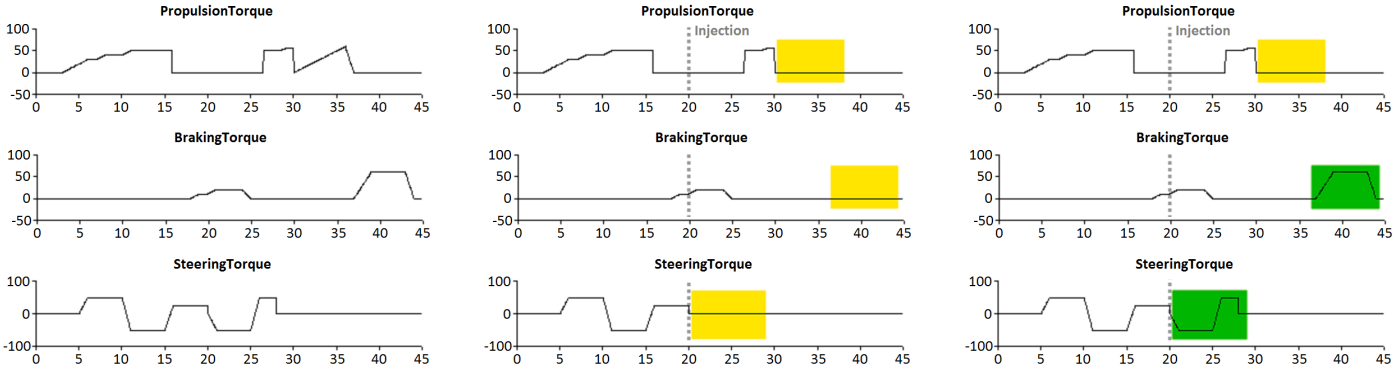
In the following, we illustrate and discuss the results of our experiments on the basis of two timing monitoring assessment scenarios.

Scenario 1: Assessing Execution Time Monitoring

In the first scenario, we evaluate the *error detection* and *error mitigation* of execution time monitoring for timing errors caused by a *permanent* hang of task *OEM_Low_40ms*. In order to inject an infinite loop in the control flow of the task *OEM_Low_40ms*, we place an injector in the runnable *ManualPropulsion*. The injection is triggered, whenever the number of invocations of the runnable passes a configurable threshold.

To enable the comprehensive observation of the system's reaction to fault injections and to analyze error propagation effects, we require access to signal traces within the system. For this reason, we have instrumented the SW-Cs *Environment* and *OEM_Hi* with 20 interceptors that are capable of logging all relevant signals within the system.

The scenario consists of three FI campaigns: (1) the fault-free golden run, (2) a series of experiments in which faults are injected at fixed, workload-dependent times and execution time monitoring is *disabled*, and (3) the same series of experiments with execution time monitoring being *enabled*. Using the flexible configuration approach of libGRINDER (cf. Section V), no re-compilation or re-flashing was necessary for the different campaigns. Instead, we used the same binary image for all three campaigns and adjusted the configuration at runtime. Each FI experiment runs for 45 seconds, which is the time during which workload stimuli are provided by the *Environment*. In the following, we discuss one experiment



(a) Fault-free *golden run*. (b) Monitoring disabled, error propagates. (c) Monitoring enabled, propagation mitigated.

Fig. 5. Scenario 1: Signal traces for a fault injection of an infinite loop (i.e., a *permanent* timing fault) in task *OEM_Low_40ms* at 20 seconds.

from each campaign in detail to evaluate the effectiveness of *error mitigation* due to execution time monitoring.

Figure 5 depicts the traces of the three signals *PropulsionTorque*, *BrakingTorque* and *SteeringTorque*. These three traces were chosen from the 20 available traces because the signals are affected by the fault injection either directly or indirectly due to error propagation. The fault-free golden run is shown in Figure 5a, whereas Figure 5b and Figure 5c depict the signals in the presence of a permanent task hang injected at 20 s where execution time monitoring is either disabled or enabled.

For the scenario in Figure 5b where monitoring is disabled, all three signals have observable deviations from the golden run at different times in their signal. The change of *PropulsionTorque* (from 30 s to 37 s) is directly accountable to the FI, as *PropulsionTorque* is a composite signal[8] that also comprises input from the *ManualPropulsion* runnable that we injected into. The change of *BrakingTorque* (from 37 s to 44 s) and *SteeringTorque* (from 20 s to 28 s) is caused by error propagation from the task *OEM_Low_40ms*, which we injected into, to the task *Tier1_Low_40ms*. *Tier1_Low_40ms* comprises the runnables *ManualBraking* and *ManualSteering* whose outputs contribute to the composite signals *BrakingTorque* and *SteeringTorque*. As direct consequence of the propagation of the timing error, critical functionality provided by both runnables is lost.

In Figure 5c the same injection scenario is depicted with execution time monitoring enabled. The monitoring mechanism correctly detects the timing error of task *OEM_Low_40ms* that is caused by the injected fault. Despite the detection of the error, the signal of *PropulsionTorque* still deviates from the golden run, as the injected fault is permanent. Therefore, the monitoring mechanism repeatedly detects the persisting timing error and, in consequence, kills the erroneous task in each period after the injection. While the chosen strategy of killing the task is inadequate to mitigate the permanent timing fault in this case, it still successfully mitigates the propagation of the timing error to other tasks. The *BrakingTorque* and *SteeringTorque* signals, which were affected by error propagation without timing monitoring in the previous scenario, are now identical to the golden run.

[8]A composite signal combines inputs from various sources.

In summary, execution time monitoring correctly detected errors due to injected timing fault in all of our experiments. Moreover, it successfully mitigated the effects of error propagation to lower priority tasks whenever monitoring was enabled, thus preventing the loss of critical functionality.

Scenario 2: Assessing Timing Monitor Interactions - Finding the Bug!

In our review of the timing monitoring implementation of a commercial AUTOSAR OS (cf. Section VI-A) we had noticed that execution time monitoring and resource lock time monitoring share a common timer to signal timing errors. If both monitoring mechanisms are enabled, the timer used by the execution time monitor is reset and overwritten by the resource lock monitor when a monitored task acquires a monitored resource. The new timer value is determined in the resource lock monitor's implementation by comparing the remaining execution time and resource lock budgets, and using the smaller value as the new budget. This ensures that budget violations, i.e., timing errors, are detected at the earliest point in time. In the error-free case, the resource is released after use and the cleanup routine of the resource lock monitor reinstates the execution time monitor. In the error case, the OS's error handling mechanisms are invoked.

In this assessment, we investigate the precedence relationship between both monitoring mechanisms with the intent to uncover any cases in which the re-activation of the original timer fails. Contrary to the previous scenario, our focus is not on the analysis of overall system behavior and error propagation effects, but on the *correctness* of the timing error detection and the *robustness* of the OS's timing monitoring mechanisms. Consequently, we do not require extensive logging mechanisms for signal traces in this scenario. Instead, we only instrument the *ProtectionHook* to provide logging of errors that are detected by the OS. Furthermore, we place injectors at two locations in the software stack. The first injector is placed between the resource acquisition and resource release in the runnable *ManualPropulsion* of task *OEM_Low_40ms* to inject arbitrary timing faults while holding a monitored resource. The second injector is placed in the resource lock monitor setup routine to inject arbitrary resource lock budget configurations. The aim of the budget injections is twofold: Firstly, we evaluate the robustness of the monitoring mechanism by injecting

unsound budget values that are normally prohibited by the GUI-based configuration (e.g., resource lock budgets that are bigger than execution time budgets). Secondly, we inject sound budget values to change the configuration of the monitoring mechanism on-the-fly during runtime, thus avoiding the time-consuming steps of reconfiguration, recompilation and flashing of the binary.

This scenario consist of one campaign, in which we inject an infinite loop (i.e., a permanent timing fault) between the acquisition and release of a resource in the runnable *ManualPropulsion*. Per experiment, one value of a series of sound and unsound resource lock budget values is injected directly in the data structures of the monitoring mechanism when the resource is acquired and the monitor is initialized.

For all of our experiments, the timing monitoring mechanisms detected that a timing error had occurred and also the point in time of the error was detected correctly. After reviewing the log data, and to our surprise, the distribution of the reported error types (execution time vs. resource lock time) did not match the distribution expected from the injected budget values. As the series of injected budget values was generated using the execution time monitor budget as median value, we expected a fifty-fifty distribution of each error type, i.e., for those experiments where the resource lock budget was smaller than the execution time budget, we expected a resource lock error, and for the opposite case an execution time error. Instead, all error types were logged as resource lock errors.

In order to verify that the observed mismatch of the distribution is not only accountable to the injection of unsound budget values, we analyzed the root cause for the mismatch further. As a result, we discovered that the type of timing error is misidentified whenever a timing error occurs while holding a locked resource and, at the same time, the remaining execution time budget is lower than the remaining resource lock budget. Whenever these constraints are met, errors accountable to execution time violations are reported as resource lock errors, affecting both, sound and unsound, configuration conditions.

This deficiency of the implementation is critical, as the handling of errors by the OS and the user (through means of the ProtectionHook) relies on the correctness of the supplied error type. A wrongfully identified error may therefore directly impact the error analysis and, in consequence, may lead to the execution of inappropriate mitigation actions. We reported the discovered issue to the vendor of the AUTOSAR OS implementation, who was able to reproduce it. The vendor acknowledged the issue as a bug and fixed it in subsequent releases of the OS.

Case Study Summary

In the case study, we have presented two scenarios for the FI-based assessment of AUTOSAR's timing monitoring mechanisms using the FI framework GRINDER. In the first scenario, we evaluated the correctness of the error detection and error mitigation of execution time monitoring for permanent timing faults. The monitor detected all injected timing faults correctly and was able to mitigate error propagation successfully, thus preserving critical functionality. In the second scenario, the correctness and robustness of execution time monitoring and resource lock monitoring was evaluated by injecting a permanent resource lock timing fault in combination with the injection of sound and unsound resource lock budgets. While the monitoring mechanisms correctly detected the presence of an error and its point in time, the source of the error was misidentified under certain conditions, which potentially affects error mitigation actions negatively. As the focus of our work was on providing the FI assessment mechanisms and guidance on their use, we only conducted around 200 experiments overall, which comprise the basis for this case study. To put this number into context, we should note that comprehensive FI studies sometimes comprise hundreds of thousands of experiments and that consequently the amount of experiments that we conducted can be considered very low. It is therefore even more impressive that, as a result of our FI experiments, we uncovered a bug in a commercial AUTOSAR OS implementation, which emphasizes and justifies the use of FI as an effective method for the assessment of AUTOSAR's safety mechanisms.

VII. Conclusion

Innovation in the automotive sector is mainly driven by software, which is leading to automotive software systems of massively increased complexity. To foster reusability, portability and interoperability of automotive software components, the AUTOSAR industry standard promotes a modular software architecture for automotive systems. At the same time, the ISO 26262 standard addresses safety considerations for automotive control systems, covering both hardware and software aspects, which has led to the addition of a safety concept to AUTOSAR comprising several safety mechanisms. While the ISO 26262 explicitly recommends fault injections (FI) to assess such mechanisms, it provides little guidance on how experiments should be performed and how suitable fault models can be identified.

In this paper we describe the process of implementing and applying FI for the validation of AUTOSAR's safety mechanisms, according to recommendations outlined by the ISO 26262 standard. To conduct the FI experiments, we have adapted the open source FI framework GRINDER to AUTOSAR. By making our implementation openly available, we provide a ready to use FI framework for AUTOSAR and hope to foster the development and use of FI for this target environment. To fill the gap between the ISO standard's requirement to apply FI and the absence of suitable fault models, we provide a detailed discussion how we derived fault models for our assessment, which targets a commercial implementation of AUTOSAR's timing monitoring safety mechanisms. The conducted experiments uncovered an actual bug in the interaction of two timing monitoring mechanisms that could lead to a misidentification of the source of a timing error and negatively impact the effectiveness of error mitigation. The bug was subsequently acknowledged and fixed by the supplier of the safety mechanism's implementation.

In summary our results demonstrate that (1) FI is an effective method to assess automotive suppliers' implementations of AUTOSAR safety mechanisms, (2) suitable fault models for these systems can be derived from their functional specification and their intended usage context, and (3) using these fault models actual deficiencies in the implementation can be identified with a modest amount of experiments.

REFERENCES

[1] S. Winter, T. Piper, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "GRINDER: On Reusability of Fault Injection Tools," in *Proc. of IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2015, to appear. [Online]. Available: http://www1.deeds.informatik.tu-darmstadt.de/External/PublicationData/1/grinder.pdf

[2] M. Broy, "Challenges in Automotive Software Engineering," in *Proc. of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 33–42.

[3] R. N. Charette, "This Car Runs on Code," *IEEE Spectrum*, Feb. 2009. [Online]. Available: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

[4] *AUTOSAR (AUTomotive Open System ARchitecture)*, Release 4.2.1, Std., Oct. 2014. [Online]. Available: http://www.autosar.org/

[5] *ISO 26262: Road vehicles – Functional safety*, International Organization for Standardization Std., 2011.

[6] AUTOSAR, *Overview of Functional Safety Measures in AUTOSAR*, AUTOSAR Release 4.2.1, Std. Document ID 664, Oct. 2014.

[7] P. Koopman, "A Case Study of Toyota Unintended Acceleration and Software Safety," Presentation, Sep. 2014. [Online]. Available: http://betterembsw.blogspot.de/2014/09/a-case-study-of-toyota-unintended.html

[8] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[9] N. Silva, R. Barbosa, J. C. Cunha, and M. Vieira, "A View on the Past and Future of Fault Injection," in *Proc. of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–2.

[10] AUTOSAR, *Technical Safety Concept Status Report*, AUTOSAR Release 4.2.1, Std. Document ID 233, Oct. 2014.

[11] P. E. Lanigan, P. Narasimhan, and T. E. Fuhrman, "Experiences with a CANoe-based Fault Injection Framework for AUTOSAR," in *Proc. of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 569–574.

[12] G. Baumgarten, M. Oertel, A. Rettberg, and G. Marcelo, "First Results of Automatic Fault-Injection in an AUTOSAR Tool-chain," in *Proc. of the 12th IEEE International Conference on Industrial Informatics (INDIN)*, 2014, pp. 170–175.

[13] L. Pintard, J.-C. Fabre, K. Kanoun, M. Leeman, and M. Roy, "Fault Injection in the Automotive Standard ISO 26262: An Initial Approach," in *Dependable Computing*, ser. Lecture Notes in Computer Science, M. Vieira and J. C. Cunha, Eds. Springer, 2013, vol. 7869, pp. 126–133.

[14] L. Pintard, J.-C. Fabre, M. Leeman, K. Kanoun, and M. Roy, "From Safety Analyses to Experimental Validation of Automotive Embedded Systems," in *Proc. of the 20th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2014, pp. 125–134.

[15] B. Vedder, T. Arts, J. Vinter, and M. Jonsson, "Combining Fault-Injection with Property-Based Testing," in *Proc. of International Workshop on Engineering Simulations for Cyber-Physical Systems (ES4CPS)*, 2014, pp. 1–8.

[16] J. M. Q. Cunha, "Fault injection for the evaluation of critical systems," Master's thesis, Universidade do Minho, 2013. [Online]. Available: http://hdl.handle.net/1822/27841

[17] R. Barbosa, N. Silva, and J. M. Cunha, "csXception®: First Steps to Provide Fault Injection for the Development of Safe Systems in Automotive Industry," in *Dependable Computing*, ser. Lecture Notes in Computer Science, M. Vieira and J. C. Cunha, Eds. Springer, 2013, vol. 7869, pp. 202–205.

[18] A. Salkham, A. Pecchia, and N. Silva, "Assessing AUTOSAR Systems Using Fault Injection," in *Proc. of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2012, pp. 11–12.

[19] ——, "Design of a CDD-Based Fault Injection Framework for AUTOSAR Systems," in *Proc. of Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, 2013.

[20] R. Chillarege and N. S. Bowen, "Understanding Large System Failures - A Fault Injection Experiment," in *Proc of the 19th International Symposium on Fault-Tolerant Computing (FTCS)*, 1989, pp. 356–363.

[21] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults Based on Field Data," in *Proc. of the 26th International Symposium on Fault-Tolerant Computing (FTCS)*, 1996, pp. 304–313.

[22] M. M. Islam, B. Sangchoolie, F. Ayatolahi, D. Skarin, J. Vinter, F. Törner, A. Käck, M. Nyberg, E. Villani, J. Haraldsson, P. Isaksson, and J. Karlsson, "Towards Benchmarking of Functional Safety in the Automotive Industry," in *Dependable Computing*, ser. Lecture Notes in Computer Science, M. Vieira and J. Cunha, Eds. Springer, 2013, vol. 7869, pp. 111–125.

[23] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A Tool for Experimental Dependability Assessment," in *Proc. of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 557–562.

[24] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "MODIFI: A MODel-Implemented Fault Injection Tool," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, E. Schoitsch, Ed. Springer, 2010, vol. 6351, pp. 210–222.

[25] M. M. Islam, N. M. Karunakaran, J. Haraldsson, F. Bernin, and J. Karlsson, "Binary-Level Fault Injection for AUTOSAR Systems," in *Proc. of the 10th European Dependable Computing Conference (EDCC)*, 2014, pp. 138–141.

[26] T. Piper, S. Winter, P. Manns, and N. Suri, "Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework," in *Proc. of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[27] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The Impact of Fault Models on Software Robustness Evaluations," in *Proc. of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 51–60.

[28] R. Natella, D. Cotroneo, J. A. Durães, and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 80–96, Jan. 2013.

[29] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, pp. 11–33, Jan. 2004.

[30] A. Johansson, N. Suri, and B. Murphy, "On the Selection of Error Model(s) For OS Robustness Evaluation," in *Proc. of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 502–511.

[31] AUTOSAR, *Description of the AUTOSAR standard errors*, AUTOSAR Release 4.2.1, Std. Document ID 377, Oct. 2014.

[32] J. A. Durães and H. S. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[33] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simFI: From Single to Simultaneous Software Fault Injections," in *Proc. of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[34] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a Versatile Fault-Injection Experiment Framework," in *Proc. of Architecture of Computing Systems (ARCS) Workshops*, 2012.

[35] A. Thomas and K. Pattabiraman, "LLFI: An Intermediate Code Level Fault Injector For Soft Computing Applications," in *Proc. of Silicon Errors in Logic System Effects (SELSE)*, 2013.

[36] J. M. Voas, "Certifying Off-the-Shelf Software Components," *Computer*, vol. 31, no. 6, pp. 53–59, Jun. 1998.

[37] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed. Springer, 2002, vol. 2304, pp. 213–228.

[38] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software," in *Proc. of the 9th European Dependable Computing Conference (EDCC)*, 2012, pp. 162–172.

[39] AUTOSAR, *Explanation of Error Handling on Application Level*, AUTOSAR Release 4.2.1, Std. Document ID 378, Oct. 2014.

[40] ——, *Specification of Operating System*, AUTOSAR Release 4.2.1, Std. Document ID 034, Oct. 2014.