ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/322686477

Virtualized-Fault Injection Testing: A Machine Learning Approach

Conference Paper · January 2018

DOI: 10.1109/ICST.2018.00037

CITATION
1

READS
248

3 authors:

Hojat Khosrowjerdi
KTH Royal Institute of Technology
9 PUBLICATIONS **45** CITATIONS
SEE PROFILE

Karl Meinke
KTH Royal Institute of Technology
65 PUBLICATIONS **473** CITATIONS
SEE PROFILE

Andreas Rasmusson
Scania
3 PUBLICATIONS **8** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Highly Adaptive Trustworthy Systems (HATS) View project

Virtues-Vinnova, FFI project View project

All content following this page was uploaded by Karl Meinke on 24 January 2018.

The user has requested enhancement of the downloaded file.

Virtualized-Fault Injection Testing: a Machine Learning Approach

Hojat Khosrowjerdi
School of Computer Science
KTH Royal Institute of Technology
SE10044 Stockholm, Sweden
Email: hojatk@kth.se

Karl Meinke
School of Computer Science
KTH Royal Institute of Technology
SE10044 Stockholm, Sweden
Email: karlm@kth.se

Andreas Rasmusson
Scania CV AB
151 87 Södertälje, Sweden
Email: andreas.rasmusson@scania.com

***Abstract*—We introduce a new methodology for virtualized fault injection testing of safety critical embedded systems. This approach fully automates the key steps of test case generation, fault injection and verdict construction. We use machine learning to reverse engineer models of the system under test. We use model checking to generate test verdicts with respect to safety requirements formalised in temporal logic. We exemplify our approach by implementing a tool chain based on integrating the QEMU hardware emulator, the GNU debugger GDB and the LBTest requirements testing tool. This tool chain is then evaluated on two industrial safety critical applications from the automotive sector.**

I. Introduction

A. Background

Fault simulation and fault injection (FI) are widely accepted techniques for assessing software robustness and error handling mechanisms of embedded systems [1]. In the automotive industry, FI has become a common practice to improve embedded systems quality and avoid the costs associated with untested safety-critical software [2]. The automotive safety standard ISO 26262 [3] recommends FI testing for electronic control unit (ECU) software at automotive software integrity levels (ASILs) C and D. Automotive recall cases such as Honda [4], Toyota [5] and Jaguar [6] show that in some extreme environmental conditions ECU hardware malfunctions can potentially give rise to ECU software faults. This highlights the importance of FI to avoid the costs and risks associated with unprotected software.

Traditionally, FI is performed late in the development process, with limited scope and coverage, e.g., hardware-in-the-loop (HIL) testing that targets only ECU pins. Furthermore, a HIL rig, which includes a mock-up of electronic and electromechanical vehicle components, is an expensive and limited resource in the industry that becomes a bottleneck during testing. Therefore, it is important to consider test automation technologies that can make fault injection testing faster and more effective, with reduced costs and times.

B. Contributions

Following a growing trend towards more agile testing by hardware virtualisation, we may ask: *can one place a virtualized HIL rig (VHIL rig) on the ECU developer's desktop and use this to perform FI testing*? This approach is an emerging reality due to modern hardware emulators such as the Quick Emulator QEMU [7], as we will show. However, VHIL testing raises two further questions: *(1) how can we automatically generate large test suites and test verdicts that exploit the additional test throughput of a VHIL rig*; *(2) how can we model safety requirements and monitor these while systematically exercising a large combinatorial set of faults*? One interesting approach to test automation that addresses both questions is *learning-based testing* (LBT) [8]. This approach to automated software testing combines: (i) *machine learning* to reverse engineer software models, and (ii) *model checking* to construct test cases that evaluate formalised safety critical software requirements. In this paper, we present a new methodology for virtualized fault injection testing by combining virtualized hardware emulation with learning-based testing. This methodology has been developed and evaluated within a collaborative research project at Scania CV.

Most of the techniques and tools proposed in the literature for FI focus on providing means and infrastructure to assist controlled occurrences of faults [9], [1], [10]. Less effort has been devoted to the problem of automating the insertion process and measuring the coverage of generated tests [11]. In some cases, FI tests are manually created by a system test engineer to cover corresponding requirements [12], [13], [14]. Therefore, FI test automation is a timely research problem. Against this background, the main contributions of this paper are:

1) a new methodology for automated FI testing of safety critical embedded systems based on formal requirements modeling, machine learning and model checking;
2) a tool chain based on integrating the QEMU hardware emulator with the GNU debugger GDB [15] and the LBTest requirements testing tool [16];
3) tool evaluation results on two industrial safety critical ECU applications, where previously unknown FI errors were successfully found.

C. Paper Overview

The organisation of this paper is as follows. In Section II we consider principles of fault injection for safety critical systems. In Section III we introduce a new methodology and tool chain for FI testing by applying learning-based testing. In Section

IV we describe the specific FI techniques supported by our tool chain. In Section V we describe two industrial FI case studies. In Section VI we evaluate the tool chain using these case studies. In Section VII, we compare our methodology to closely related approaches in the literature. Finally, in Section VIII we summarise our main conclusions and future work.

II. Fault Injection for Safety Critical Systems

In this section, we review some basic principles of fault injection, as these apply to safety critical embedded software. We also provide a short introduction to learning-based testing, which can automate both test case generation and fault injection

A. Hardware Faults in Embedded Systems

In general embedded systems can be affected by two types of faults[1] that have hardware or software roots [1]. Software faults are inherently due to design errors at various development stages. These dormant faults are latent in the code until the software is executed and the running code reaches the affected area. In other words, the behavior of a software fault is transient despite its eternal nature.

Hardware faults, on the other hand, can be categorized based on their temporal behavior (duration). **Permanent faults** have constant and everlasting effects until the fault source is removed. **Transient faults** appear only once, while **intermittent faults** lie between these two extremes. They may occur at irregular intervals, often stochastically.

Permanent (e.g., stuck-at-one/zero) and intermittent faults are mainly the consequence of component damage, wear-out or unstable hardware (due to production defects). In contrast, transient faults (also known as soft-errors [17], e.g., bit-flips) are mostly due to external disturbances such as environmental conditions, heat, radiation, electromagnetic disturbances, etc.

The importance of hardware faults in safety-critical embedded systems is not only due to their direct risk of physical damage to the system, but also due to the dependency of the corresponding safety-related software on their operation. Hence, the influence of hardware faults is usually modeled by simulation to have a better understanding of the overall consequences of such faults. For example, a simple bit-flip can cause errors of **timing** (e.g. transition delay), **data** (change of memory value) or **control-flow** (corruption of a program counter value) errors.

FI methods can be compared according to the following criteria [2]:

- *Reachability*: the ability to insert an error in the desired location.
- *Controllability*: the ability of an FI method to control *what* (fault type), *when* (time) and *where* (location) to insert faults in the reachable area.
- *Intrusiveness*: the impact of implementing an FI method on the behavior of the target system.
- *Repeatability/reproducibility*: the accuracy of an FI method to repeat single/multiple FI experiments.
- *Efficiency*: the required time and effort to conduct FI testing.
- *Effectiveness*: the ability to activate and exercise various error-handling mechanisms.
- *Observability*: the availability of means to observe and record events triggered by the fault effects.

Among the properties above, intrusiveness is one key aspect of an FI technique. More specifically, intrusiveness refers to the undesired overhead in time or space properties caused by FI instrumentation. This overhead has to be reduced to the lowest possible level to achieve accurate results comparable with real-world behaviors.

B. Learning Based Testing

Learning based testing (LBT) is a novel approach to fully automated black-box requirements testing [18]. The goals of LBT are somewhat similar to *model-based testing* [19]. However, LBT dispenses with the need to design and maintain models separately from software. Instead, active machine learning [20] is used to efficiently reverse engineer a model (most commonly a state machine) using test cases as queries. This reverse engineered model can then be subjected to model-checking techniques (see e.g. [21]) to identify safety anomalies with respect to formalised safety requirements. One approach to safety requirements modeling is to use a *temporal logic*, such as propositional linear temporal logic[2] (PLTL) [22]. Temporal modeling can be used both to specify embedded real-time software behavior, and to describe temporal fault injection patterns.

Figure 1 illustrates the architecture of the LBT tool LBTest [16]. This architecture features a feedback loop that is used to iteratively refine a learned model of the system under test (SUT) by interleaving active machine learning, model-checking, and equivalence checking queries. The initial model M_0 is a null hypothesis. Incremental learning refines each model M_i into a more detailed model M_{i+1}, which is passed to a model-checker. If any safety anomaly (i.e., execution of M_{i+1} violating a safety requirement) is found, a corresponding test case is extracted and verified against the actual SUT behavior. If the SUT behavior agrees with the behavior of M_{i+1}, then the resulting true negative is reported to the test engineer for post-mortem analysis. Otherwise, the test case is used to further refine the model M_{i+1}. For many state machine learning algorithms we can prove that the sequence $M_0, M_1, ...$ always converges [20].

In LBT, testing is usually terminated after reaching some bound on the test time or number of models. At this point, stochastic equivalence checking is used to measure the *convergence* of the final model M_{final}. To measure convergence, we estimate the capability of the final model to accurately simulate

[1] A fault is a triple consisting of a time, a place and a fault type.

[2] Propositional linear temporal logic extends propositional logic with temporal operators including *next* $X(p)$, *always* $G(p)$, and *eventually* $F(p)$. Such a logic is able to model both *safety properties* "nothing bad may happen" and *liveness properties* "something good should happen".

the SUT on random input samples using an equivalence checker (c.f. Figure 1). Thus convergence gives an indirect empirical measure of black-box test coverage. The reader is referred to [18] for a more detailed discussion of this subject.

Figure 1 also shows the communication wrapper which forms a test harness around the SUT. This user-defined software component creates a mapping between symbolic data types (used in abstract test cases) and concrete data types (used in concrete test cases). It marshals test stimuli into the SUT and returns SUT observations to the testing tool. Wrapper technology is crucial for implementing fault injection, as has been shown in [23], and this approach will be further discussed in Section III-E.

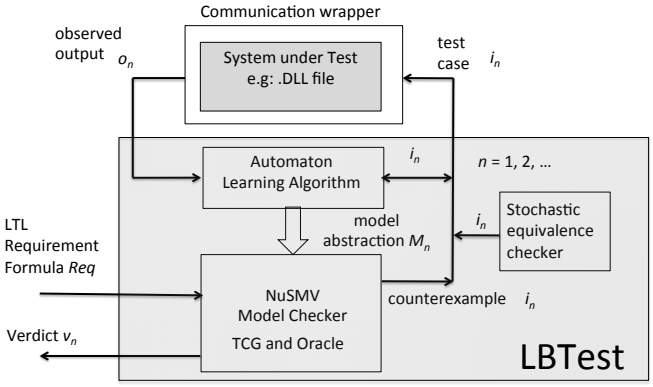

Figure 1. LBTest architecture

LBTest has been evaluated in numerous industrial case studies in different domains including avionics, finance, telecommunications, web applications and more recently automotive software e.g. [24], [23], [25], [26]. In the context of FI, [23] demonstrated satisfactory results in injecting communication infrastructure faults into a distributed microservice architecture. Here, we go beyond this early study to deal more systematically with the combinatorial growth of hardware and timing faults. Furthermore, we address the study of low-level fault propagation into high-level software components within safety critical real-time embedded software.

III. An LBT Methodology and Architecture for Fault Injection Testing

In this section, we consider a new approach to fault-injection testing of safety critical systems by combining virtualized hardware emulation with learning-based requirements testing. The goal is to support three capabilities in parallel: (i) automated test case generation, (ii) automated verdict construction based on formal requirements models, and (iii) automated fault injection.

First, we describe our approach to fault emulation based on the integration of QEMU with the GNU debugger GDB. Next, we describe an architecture that integrates QEMU-GDB with the learning-based requirements testing tool LBTest. Finally, we discuss how LBTest manages the fault injection process.

A. Fault Emulation with an integrated QEMU-GDB

The effectiveness of FI testing is greatly influenced by the plausibility of the fault models when compared with physical reality or other modeling approaches, (i.e., fault representativeness). An instruction-accurate simulator provides an abstraction of the target CPU micro-architecture that when compared to a cycle-accurate simulator abstracts away many low-level details. A fault at gate-level can affect each gate, cell or memory element; while more abstract components like multiplexers and registers are affected at the RTL level. In the case of an instruction-accurate emulator, faults are visible to state variables of each emulated component such as the PC (program counter) or processor GPRs. In other words, the faults at this level of abstraction are a subset of all possible faults at the gate or RTL levels. Thus, hardware faults whose effect directly influences the status variables can be modeled by simple soft-error faults (e.g., bit-flip and stuck-at faults). However, more advanced hardware faults with propagated outcomes can still be modeled either by combining a set of elementary fault models or by introducing extra macros to apply certain manipulations to the fault location.

Our approach to fault emulation is based on QEMU, which is a versatile open-source hardware emulation platform. It supports diverse target architectures including x86, PowerPC, and ARM. QEMU allows many unmodified guest OS to run on a host machine. It relies on a 2-step dynamic binary translation (DBT) engine to translate the target machine code into an intermediate form, optimize it to improve the performance, and then transform it to the host machine executable code using the tiny code generator (TCG). Other hardware peripherals and components are emulated using C level functions that may access the host machine resources through the OS kernel.

Hence, by the integration of QEMU and GDB, it is possible to provide a fast and realistic simulation of hardware faults. In addition, a broader window on the fault spectrum can be achieved. The overview of an QEMU-GDB architecture is depicted in Figure 2. It is equipped with a serial communication interface link to interact with generic developer tools like GDB. This interface supports inspection and manipulation of internal and boundary values of the target system (e.g., CPU registers, memory-mapped content, external and network devices, etc.). In addition, it can be used to control the simulation sequence. Therefore, it is possible to implement various fault models without SUT or QEMU intrusion.

To simulate the effect of hardware faults in embedded systems, we have considered two slightly different approaches, both relying on the Software GNU Debugger [15]. GDB is used for debugging application-level programs of embedded systems. The first approach, named SW-GDB, provides various software breakpoint types together with commands to manipulate program data at run-time and control the execution. Our solution utilizes these capabilities to introduce a fault at a particular location within the ECU software and catch anomalies in the behavior. Such defects can be precisely determined from user requirement models after integrating the

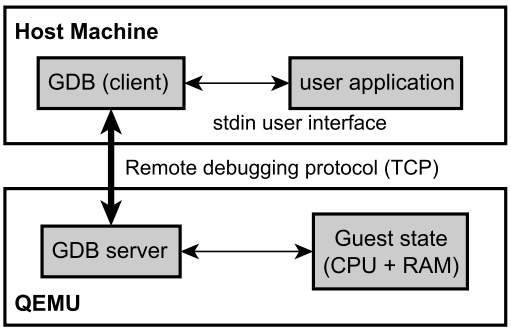

Figure 2. The QEMU-GDB architecture to emulate fault effects using user interface breakpoints

requirements testing tool LBTest.

The other technique, named QEMU-GDB, exploits the capabilities of the QEMU emulator in observing a processor's address lines and emulating hardware breakpoints as a mechanism for memory access tracing, data profiling and data race detection in a kernel [27], [28].

B. SW-GDB versus QEMU-GDB

Depending on whether we inject a fault via SW-GDB or QEMU-GDB, we can emulate faulty behavior on two different abstraction levels. Using SW-GDB, fault injection is done via data manipulation of source code variables as well as memory-mapped locations. In fact, the augmented debug information to the compiled image of the SUT provides the possibility to insert breakpoints on any line of the source code and access any software level data structures. Hence, even complex data (e.g., a multi-dimensional pointer to an array) can be observed at any desired time to inject a fault and analyze the corresponding software response. In addition, this method is not limited to a specific processor architecture due to the availability of GDB for most commonly used processors. Therefore SW-GDB can be used without intrusion into the SUT. Despite these advantages, the SW-GDB method also has weaknesses such as longer time overhead due to its trap and exception handling mechanisms [15].

QEMU-GDB works at a lower level that is closer to the hardware level simulation. Here, a fault can be injected by setting a memory access breakpoint (hardware watchpoint). Thus, we have the flexibility to control fault types, fault locations and fault insertion times dynamically without recompilation of QEMU or the SUT code. However, the SUT is more like a black-box for the QEMU-GDB method so that it cannot observe any software level data structures. Even though this black-box view will limit the controllability properties of the FI method to some extent, there still exist other accessible parameters within the hardware domain (e.g., CPU clock) to compensate these weaknesses. In addition, QEMU-GDB offers broader fault type coverage, including not only soft errors but also timing delay errors.

C. The Full FI Architecture

As illustrated in Figure 3, the architecture of our proposed approach includes five main components: (1) LBTest acting as the FI manager, test case and verdict generator. This writes test output files. (2) A communication wrapper acting as the test harness (an SUT specific python test and fault injector). (3) The FI-module that executes the code to inject faults (a GDB python script), (4) The LBTest configuration file, which defines the overall test session. (5) The GDB interface (GDB API) and the QEMU terminal environment.

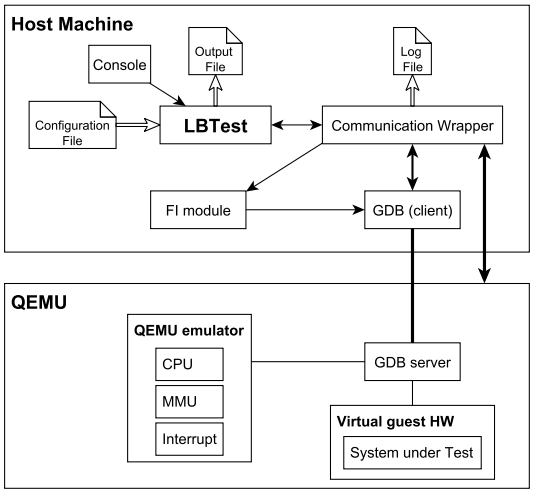

Figure 3. The integrated LBTest-QEMU-GDB tool chain with debugging from the GDB user interface

LBTest functions as both the test session and FI manager. It supplies five main functionalities: (i) generating test cases for fault detection, (ii) executing the communication wrapper, (iii) sending the test stimuli and observing the SUT behavior, (iv) reverse engineering the SUT behavioral model by machine learning, and (v) constructing test verdicts according to observed SUT behaviours and formal safety requirements by model checking.

The test engineer can define different test and FI parameters in a configuration file which is then read by LBTest to control the FI campaign. The configuration data also includes test session meta-data such as data models specific to the SUT API, machine learning, and model-checking options. Most importantly, safety critical test requirements, fault types and fault values are specified in the configuration file for use in test case generation and test verdict construction.

Symbolic fault values are scheduled into each abstract test case (i.e. symbolic input data sequence) by the test case generation processes of LBTest. These symbolic fault values are concretised into fault injection events by the FI-module module (see Section III-E below). A key feature of LBTest is platform independence from the SUT and test environment due to the use of abstract (symbolic) test data models and symbolic model checking. Hence, LBTest can support fault injection at different abstraction levels regardless of the OS.

Before the communication wrapper interacts with GDB and QEMU via their serial communication API, it has to set the corresponding FI software or QEMU level breakpoints. This is done by the FI-module so that fixed or variable FI functions are defined in GDB to activate/deactivate breakpoints depending on each test case.

The FI-module is a user application which is called inside the GDB process standard-input (stdin) by the communication wrapper. It configures the SUT based on the received arguments, sets those data or program line breakpoints that are chosen by the user, and defines optional GDB functions to be able to set on-the-fly breakpoints. With this flexibility, it extends FI test coverage to include more fault types and locations based on the input parameters it receives. Finally, the module has responsibility for exception handling and crash detection to report the appropriate response to the communication wrapper.

As shown in Figure 3, the QEMU module which was introduced in Section III-A, runs on top of the host machine's operating system. It emulates the hardware platform of the SUT in two ways. It either translates the guest processor commands into the host CPU commands or runs functions that simulate the behavior of hardware peripherals. The test harness loads the binary image of the SUT into the QEMU core and selects the target platform and the virtualization mode in which the SUT should be executed. The QEMU module is configured to receive a halt signal and wait for GDB client to connect and control its execution.

The GDB module provides methods and functions to run/stop a program, access its internal data and make changes to observe the effects.

D. LBTest as an FI Controller

Criteria that make LBTest an appealing tool choice for FI are: (i) a high degree of test automation, (ii) a high test throughput that can match the low latency of real-time embedded applications, (iii) the ability to systematically explore combinations of fault types, locations and times as test session parameters. To achieve the latter, LBTest supports useful combinatorial test techniques such as n-wise testing [29]. These are necessary to handle the combinatorial growth of both the SUT state space and the (time dependent) hardware fault space. Other important features of LBTest include scalability to large distributed systems as well as precise verdict construction from behavioral requirements. The latter is essential for FI testing the robustness of safety-critical embedded systems.

E. The Fault Injection Test Harness

Traditionally, an LBTest communication wrapper functions as a test harness to encapsulate the SUT and its environment. It marshals each test stimulus and the corresponding SUT response back and forth. It also performs test set-up and tear-down activities necessary to isolate each test experiment. Alongside these basic functionalities, the wrapper has more sophisticated responsibilities, including the following.

1) The wrapper abstracts infinite state systems into finite state models. This abstraction is done through data partitioning.
2) Wrappers support real-time abstractions using synchronous (clock-based) or asynchronous (event-based) SUT response sampling methods.
3) The semantics of FI is implemented by using the wrapper to control fault injection and simulate desired failures (e.g., a sensor or communication failure). This principle is the gateway to robustness testing using LBT [23].

In our approach to FI, the wrapper is the second component of the FI chain. It calls QEMU and GDB in separate processes and instantiates their serial communication links (using standard-input/output). Through these links, the wrapper can send GDB/QEMU commands to configure test setup, start/stop the execution of the SUT, inject faults, wait for results, monitor variables and perform post-processing on the desired outputs.

The symbolic fault values within the test cases generated by LBTest are translated into GDB executable queries. These commands, in turn, implement the fault semantics according to their specified attributes. Hence, each test case can assign its combination of fault parameters (test case fault values assigned to fault variables at specific times). Since our method is based on breakpoints to determine FI points (i.e., places where injection should occur), they are defined during test setup using the FI-module but are held deactivated before executing the SUT. The wrapper can then enable/disable the corresponding breakpoints according to every fault parameters.

IV. Supported Fault Injection Techniques

Depending on whether we inject a fault via SW-GDB or QEMU-GDB, we can emulate faulty behavior on two different abstraction levels. As mentioned previously, faults can be characterised by three attributes (*space, type, duration*). To fully describe the SW-GDB or QEMU-GDB interfaces to our tool chain, we should further specify the supported scope of these attributes in each case.

A. Fault Triples

Each injection experiment is performed based on three main attributes.

- *Fault Space*: Fault space is a set of fault points where each point represents a unique pair including time and location information. In other words, the time is distinguished by the event which triggers the fault, and the location is the target object which the fault will affect.
- *Fault Type*: A fault type determines the way data or state is corrupted. Common hardware fault types are stuck-at, bit-flip, transition delay, etc.
- *Fault Duration*: As presented in Section II-A, the lifetime of a fault is defined by the duration or the corresponding statistical distribution with which it repeats.

B. Fault Injection via SW-GDB

1) *Fault Space*: It is relatively easy to define fault points here since SW-GDB uploads the symbol information of

a program to trace its execution. This way, we can set breakpoints at any line of the source code and inject a fault into that line by manipulating accessible variable values. Furthermore, we can set conditional breakpoints which brings more flexibility in dealing with specific fault cases.

2) *Fault Type*: The GDB-Python integrated API provides the capability to perform any manipulation function at a fault injection point. With this in mind, simple (e.g., bit inversion, bit stuck-at-one/zero) and more complex (e.g., transition delay, communication) fault types can be implemented on variables, registers, and memory locations.
3) *Fault Duration*: It is possible to implement permanent and short-interval intermittent faults in SW-GDB but at the expense of a considerable increase in testing time. The main reason is the overhead caused by breakpoint artifacts (e.g., setting a trap instruction, capturing the corresponding exception, stopping program execution, saving local variables, data manipulation, etc.). However, transient and low-frequency intermittent faults are more appropriate for this approach.

C. Fault Injection via QEMU-GDB

1) *Fault Space*: The QEMU-GDB method is restricted to the hardware state information available in QEMU (e.g. registers, memory, IO) as well as function start points of the program. Hence, the time and location parameters of an FI experiment can be set, for example, based on the relative distance from a function start execution point. Another option is to set a hardware access breakpoint (access watchpoint) and perform the desired manipulation of data. This approach has the flexibility to limit the hit rate with a condition and emulate the faulty behavior of a hardware component during a particular time window.
2) *Fault Type*: After reaching an FI point, the target data is manipulated so that the desired fault can be emulated. Using data masks and custom functions, common fault types such as bit-flips and stuck-at can be implemented. Furthermore, we can adjust delays while executing QEMU hardware commands by changing the number of cycles each atomic command will take.
3) *Fault Duration*: In QEMU-GDB, hardware breakpoints are treated with very low overhead compared to SW-GDB, since they are handled as exception events. Also, implementing data masks here is usually more convenient based on the simplicity of the available data structures. Thus, in addition to transient and permanent faults, high-frequency intermittent faults can be implemented.

V. Case Studies: Validation

A. Simulation and Test Setup

The FI approach presented in Sections IV and IV has been implemented as a tool-chain on an x64-based system with the following specifications: The host machine has an i7-6820HQ CPU with 16GB installed memory. The kernel and operating system are Linux Ubuntu 3.19.0-15-generic and Ubuntu 15.04 respectively. The hard drive is a standard 500GB SATA SCSI disk.

QEMU version 2.6.0 was used as the virtualization platform[1]. The SUT hardware model implemented in C included an adaption of one of the board-models bundled with QEMU. To be able to run the ECU software without modification, additional target specific hardware models were also deployed. Furthermore, SW-GDB and QEMU-GDB versions 7.9.1 and 7.9 [15] were utilized. We used LBTest version 2.2.2 [16] which provided two learning algorithms (L^*-Mealy and *Min-Split*) for reverse engineering state machine models.

The choices of requirements modeling language and the model checker were propositional linear temporal logic (PLTL) [22] and NuSMV 2.5.6 [30] respectively. The SUT wrapper was scripted in Python 2.7.9. One motivation behind this was the availability of a Python-API for GDB as our FI solution relies on it.

Noteworthy here is that the FI-module was implemented in a script file that combines GDB and Python console commands which configure test setup and fault breakpoints.

Using this tool chain, we conducted fault injection testing of two automotive ECU applications supplied by Scania CV that were in production[2]. The first case study concerned the application layer of the ECU software which was more appropriate for the SW-GDB method. The second case study was related to a memory protection process in the kernel level where QEMU-GDB was more applicable.

B. Case Study 1: Remote Engine Start

The remote engine start application (ESTA) is a production component that is installed on many Scania trucks and buses. It provides a safe and robust engine start request from outside the driver cabin. It had been previously tested against black-box safety requirements [25]. Three goals of FI testing for ESTA were to evaluate: (1) the SUT robustness, (2) the SW-GDB method, and (3) the efficiency of LBT in identifying any safety anomalies.

1) ESTA Fault Experiment Design: To evaluate the SW-GDB method, we took a grey-box testing approach, by opening up the ESTA state machine implementation to the LBTest wrapper and injecting faults into the internal state machine variables of ESTA.

Each LBTest generated test case tc consisted of a finite sequence $tc = tc_1, ..., tc_\lambda$ of input vectors, where λ was a dynamically chosen test case length. Each input vector $tc_i = (switchVal, engineVal, pbrakeVal, faultVal)$ was a fixed length vector of values for the start switch, engine, and parking brake variables and a single fault activation variable (with values $active$ or $idle$). In comparison with [25], we considered only the bit-flip fault type. Therefore, all input signals were boolean variables, yielding $2^3 = 8$ symbolic input values. Note

[1]QEMU is distributed from https://github.com/qemu/qemu.

[2]Some technical details are omitted here for commercial reasons.

	PLTL Requirement	Type	FI Verdict
1	`G(!(ACond=SNAL & request=SRQ))`	Safety	Pass
2	`G(switch=sw_0 & X(ACond=SAL & switch=sw_1) -> F(request=SRQ))`	Liveness	Warning
3	`G(switch=sw_0 & X(switch=sw_1 & ACond=SNAL) -> F(failed=sfa))`	Liveness	Warning
4	`G(((switch=sw_1 & vtime=vtlow & ACond=SAL) & X(switch=sw_0 | ACond=SNAL)) -> F(abort=SAB))`	Liveness	Warning
5	`G(!(approved=SAP & (vtime=vtlow))`	Safety	Fail

Table I
ESTA FORMAL REQUIREMENTS USED IN FI TESTING

that $faultVal$ is the fault activation value destined for the FI-module, and not an SUT input value. The SUT wrapper loaded the QEMU and SW-GDB environment processes, and read the FI-module to set the fault injection breakpoints. The SUT sampling and FI were done synchronously every 10 milliseconds when the ESTA function breakpoint was hit. At the next breakpoint, the outputs were observed and returned to LBTest for machine learning.

Table I lists five tested requirements expressed in PLTL. Two of these are safety requirements (Requirements 1 and 5), and the rest are liveness requirements. These requirements covered the start request, fail, abort and approved scenarios as described in [25]. For example, Requirement 1 states that a start request shall not be made if the acceptance conditions (the safety criteria `ACond`) are not fulfilled. We wanted to investigate whether the injected errors in the ESTA internal state variable could violate these safety critical requirements.

C. Case Study 2: Protected Memory Checksum

The purpose of a protected memory area is to preserve configuration parameter values that are set during ECU startup and make them read-only. This is done through a series of function calls, mainly to a "simple checksum checker" after the target area has been previously initialized and locked. If an unauthorized application modifies the content of the protected area during runtime, it will be detected and sent to the handler for possible actions including ECU reset. Thus one FI testing goal was to examine how robust memory protected kernel processes are in the presence of injected hardware faults.

For FI testing, one safety critical requirement concerned scheduling tables containing pointers to all time-looped functions. These are checksummed and protected by a Scania proprietary kernel during runtime. Any data corruption in this area can lead to a hazardous vehicle state. Current best practice is to handle the situation by an ECU reset and investigate the cause afterwards. Hence, one requirement is that the corruption handler should always be able to identify unspecified manipulation of data and perform a safe reset.

1) SMCD Fault Experiment Design: The expected behavior of the *scheduled memory corruption detection* (SMCD) is shown in Figure 4-a. There are two predefined states: S_0 represents the normal operating state where no fault occurs in any scheduling table cells. However, the state is changed to S_1 when at least one cell is corrupted. Then, SMCD will trigger an ECU reset to reinitialize everything and fix potential errors. This safety requirement, expressed in PLTL was:

```
G(L1=f1 | .. | L10=f1
      -> X(state=S1 & ECU_reset = SMCD))
```

Note that the above requirement does not have any assumption about other kernel exception handlers that might trigger an ECU reset. We call these *process2* in Figure 4-b and will explain this further in Section VI.

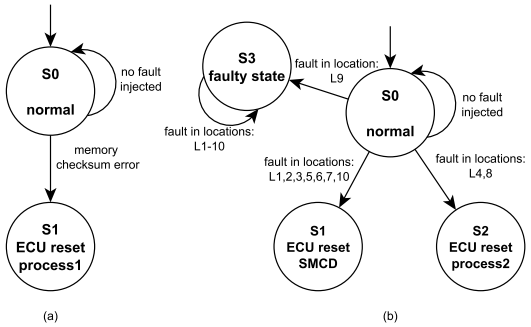

Figure 4. Protected memory checksum: intended model (a) and learned model (b)

To conduct an SMCD fault injection experiment, we needed to identify fault locations, injection times and fault types. These were defined symbolically using the LBTest configuration file, from where they were fetched and used in test case generation. We selected ten memory locations among the scheduled table cells at random. Each cell $L_1, ..., L_{10}$ was defined as a separate input variable in the LBTest configuration file with a nominal and faulty symbolic value. Faulty values caused the SUT wrapper to corrupt the content of the actual memory addresses according to the type of fault. LBTest can handle both single and multipoint faults using n-wise combinatorial testing. Hence, not only combinations of fault locations were examined, but also the fault type combinations were tested.

A fault injection time can be represented as a function or data access breakpoint in QEMU-GDB that is defined in the FI-module to be handled by the wrapper. According to Table II, three different injection times were considered. *Pre ECU init* indicates the injection occurs before the ECU initialization functions are executed. *Post ECU init* is when the initialization phase is finished, and all configuration parameters including scheduled tables are set. *Periodic loop* corresponds to the interval after the main loop starts executing and all the periodic tasks are scheduled and executed according to their timings.

In the SMCD experiments, both bit-flip and stuck-at faults were investigated. Similarly, permanent, transient and intermittent frequencies were covered. Fault types were introduced as symbolic values in the configuration file and their behaviors were implemented in the SUT wrapper. Regarding fault frequency, this was automatically controlled by the test

case generation processes of LBTest (i.e. its active learning, equivalence checking and model checking algorithms).

VI. Experimental Results and discussion

In this section, we present the results of executing the FI case studies described in Section V on the QEMU-GDB-LBTest toolchain. Our aim is to evaluate the tool chain capabilities with respect to **Error Discovery**, **Model Learning**, and **Efficiency and Performance**.

A. ESTA Bit-flip Fault Experiment

Integrating LBTest and SW-GDB is an efficient approach to simulate single event upsets (SEUs) such as bit-flip.

1) Error Discovery: As shown in the verdict column of Table I, Requirement 1 received a pass verdict from the tool. Thus, a start request was never asserted without all security conditions being satisfied. However, Requirement 5 was falsified in the fifth iteration within the learned model M_5 (size 30 states). This anomaly was not deemed hazardous since it defines an output signal to the user of a successful engine start. Nevertheless, in the ESTA logic this output was directly dependent on the fault site, and thus could be corrupted by FI.

Testing the liveness Requirements 2, 3 and 4 led to three warning verdicts, that revealed two true negatives (Requirements 2, 3) during post-mortem analysis. In these cases, the behaviors were inconsistent with the requirements on the SUT. Finally, Requirement 4 was never observed to fail. This could be because its complex precondition (the left-hand side of the implication), when tested in isolation, was never seen to pass.

The experiments using ESTA were intended as a proof of concept for evaluating the tool-chain. They revealed many inherent challenges. A significant hurdle is validating the correctness of the formal PLTL requirements themselves. This depends ultimately on having a mature hazard analysis and safety certification process in place. Emerging safety standards such as ISO 26262 can be expected to raise maturity levels in this area.

2) Model Learning: No significant differences in the learned model structure were observed when compared to the case study [25]. The injected faults slightly modified the final model with unusual transitions for some states. The ESTA model size M_6 was 30 states and 330 transitions, after 6 learning iterations. Using the SW-GDB method, each testing session took about 70 minutes on the host machine. During that time around 200 queries (6.6 per state) were executed, of which 94% were generated by active learning.

The final model convergence was somewhat low, at around 40%. This decreased to 30% for Requirement 5. However, every single path through the informal use case model was covered by at least one trace through the final learned model[1]. Furthermore, within this relatively short testing time frame true negatives were nevertheless identified.

[1]These use cases were specified in the ESTA technical documents in natural language.

3) Efficiency and Performance: The latency time of the ESTA case study was relatively long, on average 10 seconds per test case. The primary performance bottleneck was the overhead due to GDB breakpoint handling mechanisms. Therefore, through runtime monitoring, we estimated the total virtual testing time. QEMU is not cycle accurate and its main goal is to provide a fast emulation environment. Therefore, the exact timing is reliant on many unpredictable parameters, including the frequency of cache misses. The breakpoint hit rate also has an influence on the simulation speed, since SW-GDB first locates the breakpoint position by watching the program counter and then re-executes the SW to record relevant program arguments.

For example, the time it took for QEMU to emulate 10 ms (milliseconds) of the guest target clock was on average 30 ms (a slowdown rate of 3x), considering that the SW-GDB had already some significant breakpoints to avoid unimplemented hardware model exceptions. Adding the necessary SW-GDB breakpoints to perform the ESTA FI experiments, a further retardation of simulation speed to 350 ms (35x slowdown) was observed. Considering these facts, the average testing time of 70 minutes corresponded to just 2 minutes of virtual machine execution time. The performance penalty fluctuates depending on the number of information retrieval queries conducted during a breakpoint as these require communication to acquire debugging data. Even though the breakpoint overhead of SW-GDB is very high, it is still affordable for modeling bit-flips (transient faults). Its non-intrusive method implies no extra manipulation of the embedded software or QEMU (the emulating platform). The high slowdown rate suggests that parallel test case execution on multi-core platforms might be needed to speed up this approach (see e.g. [31]).

B. SMCD Fault Experiments

The safety requirement described in Section V-C1 was tested in a series of three experiments. Each experiment had slightly different FI timing, fault types and fault combinations.

1) Error Discovery: In the first experiment, only single faults were assumed to occur, and for simplicity a single injection time was considered (i.e. Post ECU init). The final learned model of Figure 4-b showed a surprisingly different state machine structure to what was expected, namely the left hand side model in Figure 4-a. LBTest reported failed test verdicts (violating the safety requirement) for three injected faults in the locations L_4, L_8 and L_9. The first two faults (L_4, L_8) were not recognized by the intended memory corruption detection (SMCD) even though they were detected by a different ECU exception handler (namely process2) that was not specified in the safety requirement. This means the SMCD process (according to the requirement) failed to handle the memory corruption injected by the tool. The last fault L_9 was not discovered by any exception handler or memory protection process at all. Therefore, it led to the faulty state S_3 where the ECU continued its operation with dramatically reduced speed. This was a single point of failure (SPOF) since it behaved like

Fault Location		Fault Injection Time					
		Pre ECU init		Post ECU init		Periodic loop	
		Bitflip	Stuckat	Bitflip	Stuckat	Bitflip	Stuckat
	L1	S1	S1	S1	S1	S1	S1
	L2	S1	S1	S1	S1	S1	S1
	L3	S1	S1	S1	S1	S1	S1
	L4	S2	S2	S2	S2	S2	S2
	L5	S1	S1	S1	S1	S1	S1
	L6	S1	S1	S1	S1	S1	S1
	L7	S1	S1	S1	S1	S1	S1
	L8	S2	S2	S2	S2	S2	S2
	L9	S3	S3	S3	S1	S1	S1
	L10	S1	S1	S1	S1	S1	S1

Table II
SMCD FI RESULTS WITH DIFFERENT TIMING AND FAULT TYPES

the SMCD functionality was disabled and it did not react to further injections in any other defined locations.

Further investigations revealed more detailed information about the SUT behavior under the fault conditions. Table II summarizes FI aspects of the case study with respect to the fault location, injection time and fault type. One key finding here was that FI for the fault locations L_4, L_8 could not be disclosed by SMCD irrespective of the fault type and injection time. Even when combined with other fault locations, in the third experiment (where we used a pairwise testing approach of location combinations), only handler process2 could identify the exception and perform an ECU reset.

Another key finding from these experiments was that the fault behavior of the SUT concerning L_9 subordinated to the injection time as well as the fault type. For example, the same bit-flip fault could lead to the safe state S_1 or the faulty state S_3 depending on whether the fault was injected before or after the ECU initialization phase was finished. Also for the stuck-at fault, the SUT is less vulnerable especially during the *post ECU init* phase and afterward.

2) Model Learning: With regard to the SMCD behavioral model, LBTest learned an average of 5 states and 100 transitions after 1.5 hours of testing. The largest learned model had 8 states and around 670 transitions. The total number of test queries tc generated by LBTest ranged between 300 and 600 of which more than 90% were generated by the active learning algorithm. The rest were divided between the model-checker and the equivalence checker. The average test case length was $\lambda = 9.8$. The SMCD models reached maximum convergence values of 74%, 67%, 44% and 20% for various combinations of fault types and locations.

With regard to the requirement error, NuSMV reported corruptions early in the testing process, a learned model with just 3 states sufficed. No false negatives among the generated counter-examples were observed, indicating the high reliability of the testing approach.

3) Efficiency and Performance: To measure the overhead implied by the QEMU-GDB hardware breakpoints, the system function *getrusage* was invoked inside the breakpoints to provide the consumed time resources by the process and its children. Whenever a hardware breakpoint was hit, the corresponding amount of time spent by the CPU in user mode and system mode were determined and accumulated as a total time value. Hence, the QEMU-GDB latency per single hardware breakpoint was estimated from averaging this figure over the number of hits. At its peak, the CPU usage time was roughly around 23.6 ms, which was 14.8 times faster than the SW-GDB approach. However, this improvement does not compensate the greater accessibility features of the first method.

VII. RELATED WORK

Various approaches to fault injection have been proposed in the literature over the years according to four main categories: *hardware-based (HFI)*, *emulation-based (EFI)*, *software-implemented (SWIFI)* and *simulation-based (SFI)* fault injection. More recently however, *model-implemented (MIFI)* and *virtualization-based (VFI)* techniques have been introduced. The literature on FI is now very extensive, and for a concise survey, we will confine our attention to the most closely related research in VFI techniques, including research on QEMU.

A. Virtualization-based Approaches (VFI)

SFI approaches that rely on instruction set simulators (ISS) and virtualization platforms are termed here VFI methods. These can also include software emulators exploiting dynamic binary translation (DBT) [7] to mimic the response of micro-architecture systems. The great flexibility and advantageous features of VFI are well suited for early development analysis and testing without an actual physical system yet existing. Some advantages of VFI include: minimized modifications, simplified test management and improved data collection. Sandboxing can be used to isolate the host environment.

The connections between system virtualization and FI testing have been a fruitful line of research in recent years especially since UMLinux was introduced in [32]. This approach is dedicated to the Linux OS to exercise dependability of networked VMs in the presence of faults. Another notable contribution is presented in [33]. This work is based on FAU-machine which is a more generic OS independent solution. Transient, intermittent and permanent faults are supported. These can be injected into memory, disk, network and other peripheral components. Thanks to its virtualization technology, FAUMachine presents high-speed simulation for complex embedded systems. In addition, it supports scripting via VHDL commands to define fault specifications and automate testing. One weakness in this approach is the lack of FI support for CPU registers (limited reachability).

More recently, a few approaches based on LLVM (Low Level Virtual Machine) have also been proposed [34]. LLVM as described in [10], is a compiler framework with additional support for program and transformation analysis between various steps of compilation and build times. It uses an intermediate representation to symbolically represent program code for the LLVM optimizer, and therefore host mid-level transformation and analysis events. Since LLVM is independent of the source language and the target machine, its derived FI tools also inherit the same properties. For example, LLFI

[35] and KULFI [36] are developed such that faults can be injected into an intermediate level of the application source code. The advantage of this method is that fault effects can be conveniently traced back to program points. However, no hardware fault modeling is permitted since the consequence of many hardware faults (e.g., those affecting registers and control flow) are not visible at the application layer (limited reachability and controllability).

In [37] an FI technique is presented relying on the Xen virtual machine monitor (VMM). This approach can inject faults into kernel or VMM level code, memory, or registers of so called para-virtualized (PV) and fully-virtualized (FV) VMs. Hence, non-virtualized systems can be characterized, and performance and fault isolation can be evaluated among VMs and VMM under faulty conditions. One concern in this method is whether the logging mechanism's overhead can corrupt the fidelity of the injection results.

Among commercial products, Relyzer [38] and CriticalFault [39] are Simics based FI tools. To reduce FI experiments, the former uses equivalence-based pruning and the latter vulnerability analysis. With these static and dynamic techniques, application instructions are grouped into various equivalence classes including (among others) control, data and address instructions. Then soft errors are injected into different points to profile faulty behavior or detect SDCs (silent data corruptions).

B. QEMU-based Fault Injection

There is considerable prior work on QEMU-based FI frameworks in the literature. QEFI [40] is one of the first frameworks designed to examine system and kernel level robustness in the presence of faults. This work instruments TCG and integrates the GDB to introduce errors at injection points. The injection locations include CPU, RAM, USB and some other external devices. In addition, faults can be triggered based on a user defined probability model, breakpoints or device activity measures. However, the fault model is only a combination of soft errors (e.g., bit-flips) and disturbances in packet communications. Moreover, QEFI is specifically designed for the Linux kernel and does not provide any support for x86 architectures. Another soft error injection approach based on QEMU is presented in [17] with the aim to evaluate the susceptibility of complex systems to certain fault types. However, this approach is limited to x86 architectures, and only bit-wise fault models on GPRs (general purpose registers) are considered. A more flexible approach is F-SEFI [41] which targets HPC (high performance computing) applications that are susceptible to silent data corruption (SDC). F-SEFI provides a broadly tunable injection granularity from the entire application space to specific functions. One shortcoming of the approach is that it only partially covers fault locations, limited to processor model variables, by tainting register values during operand execution.

A number of works have focused on mutation testing for embedded systems. XEMU [42] is an extension of QEMU integrated into its DBT with the ability to mutate disassembled binary code of an ARM instruction set at run-time. Hence, the approach does not assume access to the source code, and it can capture anomalies in the compiled target ISAs. Nevertheless, XEMU is mainly concerned with software faults, and its scope is not hardware fault injection.

Ping Xu et.al [13], [12] proposed BitVaSim with the initial aim of evaluating built in test (BIT) mechanisms in ARM and PowerPC platforms. The tool covers a variety of hardware faults not only in memory but also in I/O and CPU variables. Thus, it allows a good degree of user-mode fault configuration in any process thanks also to the XML technology. Furthermore, BitVaSim provides efficient feedback and monitoring features to control FI experiments and VM environment. One issue with this method is that it is not entirely deterministic. Hence, the reproducibility of an FI campaign is one concern.

Important work on FI by targeting the QEMU-DBT process is presented by Davide Ferraretto's team in [43], [44], [45]. In their approach, CPU GPRs and IRs are subjected to permanent, transient and intermittent faults with equivalent representativeness to RTL models. However, no memory or hardware peripheral faults are presented, and only x86 and ARM architectures are supported.

FIES, the framework described in [46], is the basis for applications in [47] and [48], where the fault models go beyond the low-level elementary bit flips and stuck at faults. Although the QEMU instrumentation remains within the DBT, more advanced memory and processor fault models are introduced to comply with IEC 61508 standard. This allows system level analysis and reliability aware software development and countermeasures for embedded and safety critical domains.

VIII. Conclusions and Future Work

We have presented a new methodology for automation of FI testing based on the integration of learning based requirements testing and hardware virtualisation. This methodology has been implemented in a specific tool chain LBTest-QEMU-GDB with the following characteristics:

- non-intrusive, i.e., no additional modification to the SUT or hardware models in QEMU is required;
- support for traditional hardware fault models, e.g., stuck-at, transient, and coupling faults at CPU registers, memory mapped components and ECU pins;
- adoption of formalized temporal logic requirements, machine learning and model-checking technology to automate test case generation, fault injection and verdict construction;
- use of conventional debugging technology for fault simulation on top of QEMU.

Our tool chain has been evaluated on two safety critical industrial case studies of ECU applications. It was successful in finding previously unknown anomalies around safety requirements.

Future research areas include multi-threaded hardware emulation on multi-core platforms to improve the performance and scalability of our technique, and multi-ECU hardware models for FI testing at integration and systems levels.

REFERENCES

[1] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer Science & Business Media, 2003, vol. 23.

[2] V. Reyes, "Virtualized Fault Injection Methods in the Context of the ISO 26262 Standard," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 5, no. 1, pp. 9–16, Apr. 2012. [Online]. Available: http://papers.sae.org/2012-01-0001/

[3] "Supporting processes, baseline 17," International Organization for Standardization, ISO 26262:(2011)—Part 8 Road vehicles—functional safety, 2011. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:26262:-8:ed-1:v1:en

[4] "Honda recalls 2.5 million vehicles on software issue," *Reuters*, Aug. 2011. [Online]. Available: https://www.reuters.com/article/us-honda-recall/honda-recalls-1-5-million-vehicles-in-united-states-idUSTRE77432120110805

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

[6] R. Charette, "Jaguar Software Issue May Cause Cruise Control to Stay On," Oct. 2011. [Online]. Available: https://spectrum.ieee.org/riskfactor/transportation/advanced-cars/jaguar-software-issue-may-cause-cruise-control-to-stay-on

[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, 2005, pp. 41–46.

[8] K. Meinke and M. A. Sindhu, "Incremental learning-based testing for reactive systems," in *Tests and Proofs: 5th International Conference, TAP 2011, Proceedings*, M. Gogolla and B. Wolff, Eds. Springer, 2011, pp. 134–151.

[9] 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.

[10] R. Piscitelli, S. Bhasin, and F. Regazzoni, "Fault attacks, injection techniques and tools for simulation," in *Hardware Security and Trust*. Springer, 2017, pp. 27–47. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-44318-8_2

[11] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "MODIFI: A MODel-Implemented Fault Injection Tool," in *Computer Safety, Reliability, and Security*. Springer, Berlin, Heidelberg, Sep. 2010, pp. 210–222. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-15651-9_16

[12] Y. Li, P. Xu, and H. Wan, "A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing," *Applied Mechanics and Materials*, vol. 347-350, pp. 580–587, 2013. [Online]. Available: https://www.scientific.net/AMM.347-350.580

[13] J. Xu and P. Xu, "The Research of Memory Fault Simulation and Fault Injection Method for BIT Software Test," in *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, Dec. 2012, pp. 718–722.

[14] A. Sung, B. Choi, W. E. Wong, and V. Debroy, "Mutant generation for embedded systems using kernel-based software and hardware fault simulation," *Information and Software Technology*, vol. 53, no. 10, pp. 1153–1164, Oct. 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584911000863

[15] "GDB Documentation." [Online]. Available: http://www.gnu.org/software/gdb/documentation/

[16] K. Meinke and M. A. Sindhu, "Lbtest: A learning-based testing tool for reactive systems," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 447–454. [Online]. Available: http://dx.doi.org/10.1109/ICST.2013.62

[17] F. d. A. Geissler, F. L. Kastensmidt, and J. E. P. Souza, "Soft error injection methodology based on QEMU software platform," in *2014 15th Latin American Test Workshop - LATW*, Mar. 2014, pp. 1–5.

[18] K. Meinke, F. Niu, and M. Sindhu, *Learning-Based Software Testing: A Tutorial*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 200–219. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34781-8_16

[19] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[20] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[21] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, Cambridge, MA, USA, Jan. 1999. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0262032708

[22] E. A. Emerson, "Temporal and modal logic." *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, vol. 995, no. 1072, p. 5, 1990.

[23] K. Meinke and P. Nycander, *Learning-Based Testing of Distributed Microservice Architectures: Correctness and Fault Injection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 3–10. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49224-6_1

[24] L. Feng, S. Lundmark, K. Meinke, F. Niu, M. A. Sindhu, and P. Y. H. Wong, *Case Studies in Learning-Based Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–179. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41707-8_11

[25] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Learning-Based Testing for Safety Critical Automotive Applications," in *Model-Based Safety and Assessment*, ser. Lecture Notes in Computer Science. Springer, Cham, Sep. 2017, pp. 197–211. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-64119-5_13

[26] K. Meinke, "Learning-Based Testing of Cyber-Physical Systems-of-Systems: A Platooning Study," in *Computer Performance Engineering*, ser. Lecture Notes in Computer Science. Springer, Cham, Sep. 2017, pp. 135–151. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-66583-2_9

[27] "Internals/Breakpoint Handling - GDB Wiki." [Online]. Available: https://sourceware.org/gdb/wiki/Internals/Breakpoint%20Handling

[28] E. Jeong, N. Lee, J. Kim, D. Kang, and S. Ha, "FIFA: A Kernel-Level Fault Injection Framework for ARM-Based Embedded Linux System," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 23–34.

[29] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2nd Edition, 2016.

[30] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364. [Online]. Available: http://dx.doi.org/10.1007/3-540-45657-0_29

[31] K. Meinke, "Learning-based testing of cyber-physical systems-of-systems: A platooning study," in *Computer Performance Engineering - 14th European Workshop, EPEW 2017, Berlin, Germany, September 7-8, 2017, Proceedings*, 2017, pp. 135–151.

[32] V. Sieh and K. Buchacker, *UMLinux - A Versatile SWIFI Tool*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 159–171. [Online]. Available: https://doi.org/10.1007/3-540-36080-8_16

[33] S. Potyra, V. Sieh, and M. D. Cin, "Evaluating fault-tolerant system designs using faumachine," in *Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, ser. EFTS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1316550.1316559

[34] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[35] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–12.

[36] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, Dec 2013, pp. 41–50.

[37] S. Winter, M. Tretter, B. Sattler, and N. Suri, "simfi: From single to simultaneous software fault injections," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013, pp. 1–12.

[38] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application resiliency analyzer for transient faults," *IEEE Micro*, vol. 33, no. 3, pp. 58–66, May 2013.

[39] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012, pp. 1–12.

[40] "QEMU-BASED FAULT INJECTION FRAMEWORK." [Online]. Available: https://www.researchgate.net/publication/260260503_QEMU-BASED_FAULT_INJECTION_FRAMEWORK

[41] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1245–1254.

[42] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "XEMU: An Efficient QEMU Based Binary Mutation Testing Framework for Embedded Software," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380368

[43] D. Ferraretto and G. Pravadelli, "Simulation-based Fault Injection with QEMU for Speeding-up Dependability Analysis of Embedded Software," *Journal of Electronic Testing*, vol. 32, no. 1, pp. 43–57, Feb. 2016. [Online]. Available: https://link.springer.com/article/10.1007/s10836-015-5555-z

[44] G. D. Guglielmo, D. Ferraretto, F. Fummi, and G. Pravadelli, "Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software," in *2013 18th IEEE European Test Symposium (ETS)*, May 2013, pp. 1–6.

[45] D. Ferraretto and G. Pravadelli, "Efficient fault injection in QEMU," in *2015 16th Latin-American Test Symposium (LATS)*, Mar. 2015, pp. 1–6.

[46] A. Höller, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, "FIES: A Fault Injection Framework for the Evaluation of Self-Tests for COTS-Based Safety-Critical Systems," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec. 2014, pp. 105–110.

[47] A. Höller, G. Macher, T. Rauter, J. Iber, and C. Kreiner, "A Virtual Fault Injection Framework for Reliability-Aware Software Development," in *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, Jun. 2015, pp. 69–74.

[48] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks," in *2015 Euromicro Conference on Digital System Design*, Aug. 2015, pp. 530–533.

View publication stats