GRINDER: On Reusability of Fault Injection Tools

Stefan Winter*, Thorsten Piper*, Oliver Schwahn*, Roberto Natella†, Neeraj Suri*, Domenico Cotroneo†
*DEEDS Group, TU Darmstadt, Darmstadt, Germany
†DIETI, Federico II University of Naples, Naples, Italy
{sw | piper | os | suri}@cs.tu-darmstadt.de, {roberto.natella | cotroneo}@unina.it

***Abstract*—Fault Injection (FI) is an established testing technique to assess the fault-tolerance of computer systems. FI tests are usually highly automated for efficiency and to prevent human error from affecting result reliability. Most existing FI automation tools have been built for a specific application domain, i.e., a certain system under test (SUT) and fault types to test the SUT against, which significantly restricts their reusability.**

To improve reusability, *generalist* fault injection tools have been developed to decouple SUT-independent functionality from SUT-specific code. Unfortunately, existing generalist tools often embed subtle and implicit assumptions about the target system that affect their reusability. Furthermore, no assessments have been conducted how much effort the SUT-specific adaptation of generalist tools entails in comparison to re-implementation from scratch. In this paper, we present GRINDER, an open-source, highly-reusable FI tool, and report on its applicability in two very different systems (the Android OS in an emulated environment, and a real-time AUTOSAR system) under four different FI scenarios.

***Index Terms*—Fault Injection, Robustness Testing, Test Automation, Test Tools, Software Reuse**

I. Introduction & Related Work

Fault injection (FI) is a commonly applied testing technique for assessing the robustness of system components (software and hardware alike) by exposing them to external defects. In order to maximize the test throughput and prevent human errors from affecting the results, FI experiments are usually highly automated by software tools and frameworks.

Although a variety of FI tools exist (cf. [1]–[4]), most of them are tailored for a specific FI technique and the targeted system under test (SUT). As a result, testers tend to develop custom tools for new SUTs, either because no FI tool exists for that SUT or the effort to identify it among the large number of existing tools (and adjust it) exceeds the effort for re-implementation.

Schirmeier et al. [5] distinguish between *generalist* and *specialist* tools. Specialists (e.g., Xception [6], FERRARI [7], Ballista [8], LFI [9], SAFE [10]) support specific SUT classes and FI techniques and are highly customized for their intended use case. On the positive side, such specialists can provide very efficient implementations with a very low runtime overhead, as they can exploit SUT-specific assumptions. Xception, for instance, makes use of special CPU debug registers to perform injections with minimal intrusiveness. On the downside, such high specialization causes a strong coupling between the FI tool and the SUT, which limits reuse of the FI tool for other experiments. In the case of Xception the applicability of minimally intrusive injections relies on the presence of the required debug registers and functions in the targeted processor and cannot be used on hardware platforms that do not feature these mechanisms.

Generalists (e.g., GOOFI [11], NFTAPE [12], FAIL* [5]) are adaptable to various SUTs and FI techniques. They provide means to reuse common modules and avoid the overhead for their re-implementation. Generalists are built around an extensible architecture with interfaces for extensions and customization to tailor them for specific use cases.

Even though a number of generalist tools have been proposed, none of them matured to a point where it could easily be applied to new classes of SUTs unforeseen by the original authors of the tools. In fact, even if generalist tools are designed for high portability and reusability, they often embed subtle and implicit assumptions about the SUT (e.g., assumptions about the execution environment and the failure modes of the SUT) that hamper their applicability for other SUTs. As a result, there is no generalist fault injection tool that is widely adopted outside the context of the research projects in which it was developed. Unfortunately, no previous experience report on the practical issues that arise when applying a generalist fault injection tool to a new system exists, thus providing little guidance for prospective users and researchers.

Paper contribution: To fill this gap, we report on our experience with GRINDER, a generalist FI tool that we implemented and that we publish under an open source license with the goal of lowering the bar for the adoption of fault injection by researchers and end-users[1]. We have made great efforts to identify and explicitly document GRINDER's interface to (and intended interactions with) the SUT. We discuss the result of these efforts and GRINDER's application in four different FI scenarios with two different SUTs: Timing error injections on the application and OS level on an AUTOSAR automotive platform (Section IV-A) and interface injections and code mutations on the Android OS (Section IV-B). Our findings indicate that a large part of GRINDER is reusable across the different application scenarios (up to 72 %). Moreover, our study highlighted a case in which we needed to sacrifice reuse to improve the performance of fault injection experiments. Nevertheless, even in this case the amount of reused code accounted for more than 54 % of the overall code for conducting the tests.

[1]https://github.com/DEEDS-TUD/GRINDER

Acknowledgments: Work supported by the DFG GRK 1362, CASED, EC-SPRIDE, EC H2020 #644579, CECRIS FP7, and SVEVIA MIUR projects

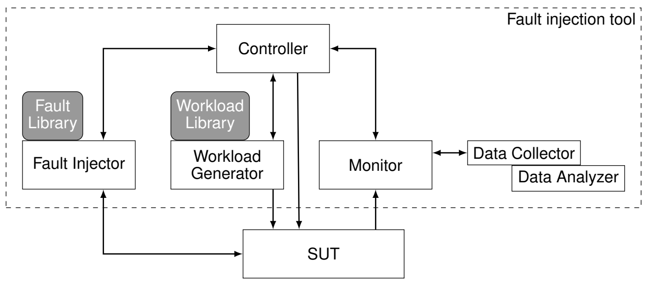

Fig. 1. Common modules of a fault injection tool. Adopted from Hsueh, Tsai, and Iyer [1] with modifications.

II. The Common Architecture of Fault Injection Tools

To illustrate how FI tests are commonly conducted and which steps in the test process are SUT-independent (and, hence, bear a potential for code reuse), we show the general structure of a fault injection tool in Figure 1 (adopted from Hsueh, Tsai, and Iyer [1] with modifications). The *Fault Injector* component performs fault injections into the SUT. It is configured and triggered by a central controller, which also triggers SUT initialization and exposes it to a workload via the *Workload Generator*. Together with the injected fault (the so-called *fault load*) the workload constitutes the test input or stimulus in FI tests. The test result is evaluated by the *Data Analyzer* based on data that the *Monitor* component collected during the test. Obviously, all components from Figure 1 that directly interact with the SUT contain at least some SUT-dependent code. SUT-independent code, like reusable fault types or workloads, can be implemented in corresponding libraries. Components that have no direct SUT-dependency (i.e., the controller, data collection, and data analysis) can be reused, unless implicit couplings with the SUT exist, e.g. data analyses on SUT-specific data. Specifically, the controller and test data processing components bear a high potential for code reuse, whereas the fault injector, the workload generator, and the monitor bear a lesser potential for code reuse, as their direct interaction with the SUT *necessarily* implies SUT-dependence to a certain degree.

III. The GRINDER Test Tool

We typically use FI tests to evaluate the robustness of components within the SUT, i.e., their ability to cope with external faults. Such external faults may originate from other components in the SUT or from the SUT's operational environment. The robustness of a component under evaluation (CUE) is inferred by monitoring its response to these perturbations.

Figure 2 illustrates an example of such a FI-based robustness assessment. The SUT consists of the CUE and two more components that interact with the CUE via interfaces A and B. To assess if the CUE can gracefully handle erroneous component behavior at interface B, we inject a fault into the component attached to interface B (e.g., using code mutation) or we perturb interactions at the interface (e.g., by corrupting parameter values or function call sequences). If the CUE is

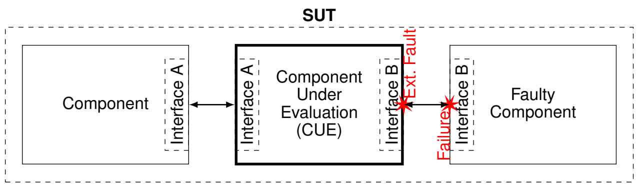

Fig. 2. Rationale of FI-based robustness tests

able to tolerate the external fault and continues to provide correct service at interfaces A and B, the component under evaluation is robust with respect to the injected fault.

Based on our experience building specialist FI tools for various SUTs [10], [13], [14] and the frequently encountered need for reimplementing previously developed functionality, we developed GRINDER (GeneRic fault INjection tool for DEpendability and Robustness assessments) as a generalist tool. We were not able to reuse our previous specialist tools on new SUTs targeted by our studies [15], [16] and existing generalists were not publically available, except for FAIL* [5].

However, FAIL* has been developed with the primary intention to emulate hardware failures and assumes the SUT to execute in an emulator. This underlying assumption on the SUT interface via an emulator renders FAIL* inapplicable for testing systems that require execution on an actual hardware platform, such as real-time systems or systems for which no appropriate emulator is available. Circumventing this constraint would have required major changes to FAIL*'s architecture, making the development of a new generalist tool the more viable option. In fact, some of our tests require execution on a hardware platform, for which no suitable emulator is available (cf. Section IV-A for details).

GRINDER is written in Java and follows the general FI tool architecture we introduced in Figure 1. Like other state-of-the-art generalists, GRINDER implements the FI experiment flow independently from a specific SUT and injection method. GRINDER provides two interfaces for SUT interactions.

- Test stimuli (faultload and workload), SUT configuration data, test logs, etc. are transmitted via a *communication channel*. Different communication channels can be utilized for different SUTs. For all of our experiments with GRINDER to date we used TCP.
- To control the SUT (start/halt it, initiate and respond to interactions via the communication channel), GRINDER requires SUT-specific code. To supply GRINDER with the required functions, testers implement a *TargetAbstraction* interface defined by GRINDER.

Figure 3 illustrates the interfaces between GRINDER and the SUT. We discuss both interfaces in the following.

A. Communication Between GRINDER and the SUT

The end points of GRINDER's communication channels within the SUT are *interceptors*. These are probes in the SUT that can be used to inject faults at SUT runtime or monitor information about the SUT's state. This separation of mechanism (interception) from functionality (injection, monitoring)

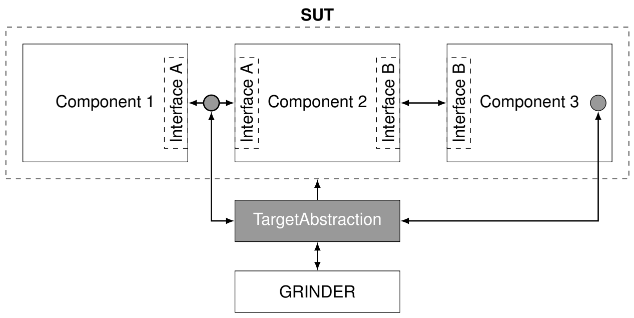

Fig. 3. Interactions between GRINDER and an instrumented SUT

makes injectors and monitors reusable at different interceptor locations in the SUT. If, for instance, debugger probes are used for interception, each such probe can be utilized to perform injections (e.g., by altering data) or to monitor the results of injections (e.g., by logging data). Interceptors are indicated in Figure 3 by gray circles at the SUT's internal inter-component interface A and inside component 3. Their connection to the FI framework is realized through the aforementioned communication channel.

Communication channels between interceptors and the FI framework are highly SUT-dependent, at least on the lower layers of the protocol stack. If the SUT is running in a virtual machine, the communication link may be a virtual network or a hypervisor-specific interface. If the SUT is an embedded control system, network or debug connectors may be used. To support diverse implementations of the communication channel in GRINDER, testers can provide corresponding implementations of interceptors and the TargetAbstraction.

The process of placing interceptors in the SUT, commonly referred to as *instrumentation*, is usually performed automatically by a separate tool that is not part of FI tool chain, due to the high dependency on implementation details of the SUT. Debugging probes, for instance, can be inserted by compilers.

B. TargetAbstraction

The TargetAbstraction interface specifies a simple set of functions that must be implemented for a SUT to be controllable by GRINDER. The specification of the TargetAbstraction was driven by the observation that on an abstract level the progression of FI experiments across different tools and SUTs is the same: SUT initialization, workload invocation, fault injection, and data collection. Consequently, the TargetAbstraction comprises these basic control functions that GRINDER requires to automatically run FI tests. The choice of these functions is based on a survey of FI-related literature (cf. [1]–[4]) and our own experience with FI experimentation.

Figure 4 shows the TargetAbstraction class that testers have to implement and GRINDER's TargetController, which directly maps the functions the TargetAbstraction provides to its control functions for conducting automated FI tests: GRINDER *starts* the SUT, *runs* a test, and stores data for offline analysis. The SUT is then *reset* to a known stable state to avoid the impact of undetected residual injection effects on subsequent tests. These steps are repeated for each test that has been added to GRINDER's test case database for the targeted SUT. After all tests from the database have been performed, the SUT is *stopped* and exchanged or reconfigured, if this is required for subsequent tests. The framework automatically stops execution after the last specified test case.

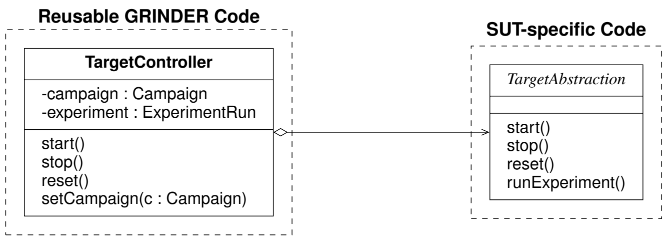

Fig. 4. The TargetController class and the TargetAbstraction interface

IV. GRINDER Case Studies

In this section, we report our experience of adapting and applying GRINDER for automated FI tests of two different SUTs: An adaptive cruise control system based on AUTOSAR [17], the de facto standard for automotive systems, and the Android OS [18]. For each of the SUTs, we implemented two different FI test scenarios to assess GRINDER's adaptability to different injection mechanisms and locations. On AUTOSAR we tested timing protection mechanisms against (Scenario 1) timing errors in applications and (Scenario 2) timing errors in the AUTOSAR OS kernel. On the Android OS we tested the OS kernel's robustness against device driver errors, a common source of OS outages [19], [20]. We injected faults into (Scenario 3) the interface between the kernel and the device drivers and (Scenario 4) the device drivers' source code.

To substantiate our generality and reusability argument for GRINDER, we focus the discussion on its adaptation to both SUTs and omit the provision of actual FI test results, which are partially documented in our publications [15], [16] and partially under submission. We assess the effort to adapt GRINDER to the different SUTs and FI tests in Section IV-C.

A. AUTOSAR SUT Setup

AUTOSAR supports the development of safety-critical systems by a set of protection mechanisms (e.g., execution time monitoring and memory partitioning) that are specified in its Technical Safety Concept [21]. To assess the *correctness* and *robustness* of automotive suppliers' implementations of AUTOSAR's timing protection, we use GRINDER to inject faults into different locations of AUTOSAR's software stack: (Scenario 1) application layer tasks and (Scenario 2) OS services. The injections aim to provoke timing errors, which the protection mechanisms should detect and mitigate.

Our SUT is an AUTOSAR-based adaptive cruise control that consists of six tasks with different degrees of criticality. Critical tasks are protected from timing interference with other tasks using AUTOSAR's timing protection mechanisms. We have implemented the test system on a Freescale XKT564L

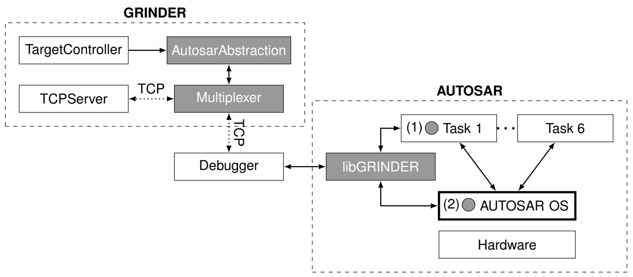

Fig. 5. Adapting GRINDER to an AUTOSAR SUT

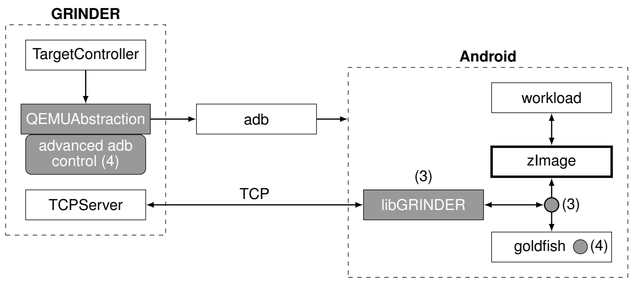

Fig. 6. Adapting GRINDER to an Android SUT

evaluation board using Elektrobit tresos Studio for system integration. The XKT564L hosts a 32-bit dual core Power Architecture microcontroller, and is connected to a host computer via a JTAG/Nexus hardware debugging interface. On the host computer, the debugger of Green Hills's MULTI IDE is used by GRINDER to interact with the hardware.

The integration of GRINDER in the AUTOSAR evaluation environment is depicted in Figure 5. Components that contain AUTOSAR-specific code, which had to be developed or adapted for GRINDER's use with the new SUT, are highlighted. The *AutosarAbstraction* implements GRINDER's TargetAbstraction interface (cf. Figure 4) and controls the evaluation board using the MULTI debugger, which GRINDER interacts with through its TCP communication channel. GRINDER's *TCPServer* provides a generic communication interface for configuration and logging, which can be used with a variety of SUTs without modification. As the XKT564L evaluation board is not equipped with an Ethernet interface for direct connection, we employ a *Multiplexer* for interactions of GRINDER's AutosarAbstraction and TCPServer with the SUT and vice versa. Within the SUT, the *libGRINDER* C library provides pre-configured injector, detector, and logging logic for the interceptors, which use an AUTOSAR-specific communication interface to exchange messages between the SUT and GRINDER through the debugger.

To avoid the time-consuming cycle of re-compilation and re-flashing between experiments with different interceptor configurations, e.g., for injecting at different fault locations or logging at different monitor locations, libGRINDER provides a flexible configuration system to selectively enable and disable interceptors at run-time, and change their configuration via parameters to account for different fault types. Consequently, the SUT requires instrumentation with interceptors only once, which allows for the uninterrupted automated execution of various test suites with diverse test case configurations.

B. Android SUT Setup

Android [18] is a popular Linux-based operating system for mobile devices. Linux device drivers have high defect densities [22] and the goal of our case study is to investigate the impact of device driver failures on Android's stability. Android's development tools ship with a QEMU-based emulator for convenient development without access to a mobile hardware platform. GRINDER runs on the same host as the emulator, controls the emulator process, and interacts with the Android system via the Android Debugging Bridge (adb).

Figure 6 gives an overview of the Android SUT. The Android kernel (*zImage*) is the component under evaluation. The *goldfish* SD card driver's misbehavior is simulated by FI. The *workload* component is an Android app that triggers kernel/driver interactions to activate injections. System instability is detected by monitoring existing Linux kernel mechanisms that signal *kernel oops/panic* errors via adb. The TargetAbstraction (*QEMUAbstraction*) is implemented by using QEMU's functions to start/stop/reset the SUT.

We have implemented two versions of the Android experiment setup with GRINDER according to Scenario 3 and Scenario 4 from the introduction of this section. In Scenario 3 we inject faults by corrupting function call parameters in the interface between the driver and the kernel. The interceptor is implemented as an interface wrapper between these components. The injection logic is contained in libGRINDER, which is built as a Linux kernel module. GRINDER first starts the emulator, then loads the interceptor after boot-up, which in turn loads the driver. GRINDER then starts the workload, which eventually triggers kernel/driver interactions, which are altered for injections by the interceptor. If an error occurs, a corresponding message (kernel oops or panic) is parsed from the adb output and the test ends. If the workload finishes without error, this is also signaled via adb. If none of these events occurs, a timeout triggers and signals a test hang. Upon any of these events, the emulator is shut down and the corresponding test result stored in GRINDER's test database.

In Scenario 4 we apply code mutations to emulate representative residual software defects [10] in the driver. Injections are performed upfront on the driver source code. The system is started, the corresponding mutant loaded, and the workload started to trigger the fault. The detection mechanisms and emulator start-up/shut-down are identical to Scenario 3.

C. Software Reuse

Table I illustrates the implementation effort for both case studies. For both test scenarios on the AUTOSAR SUT, the SUT-specific implementations of the AutosarAbstraction, the interceptors, and the communication channel accounted for $523\,\mathrm{SLOC}$ (source lines of code[2]), which is less than $18\,\%$

[2]All SLOC counts were generated using David A. Wheeler's SLOCCount.

TABLE I
IMPLEMENTATION EFFORT IN SLOC FOR THREE DIFFERENT FI ASSESSMENTS

Case Study	Reusable SLOC (GRINDER)	SUT specific SLOC	Scenario specific SLOC	Total
AUTOSAR (Scenario 1) & (Scenario 2)	2429	523	–	2952
Android (Scenario 3)	2429	–	1121	3550
Android (specialist)	–	–	2464	2464
Android (Scenario 4)	2429	–	2028	4457

of the overall framework implementation for this case study. As the SUT-specific code in libGRINDER was reused for both scenarios (the main difference was the instrumentation that is not covered by the FI tool), there is no scenario-specific code. For the Android OS assessment (Scenario 3), the scenario-specific code accounted for $1121\,\mathrm{SLOC}$, more than $31\,\%$ of the overall code. The main reason for this difference is the diversity of the intercepted interfaces. While for the AUTOSAR example the interceptor logic was simple and mostly reusable across different experiments, the Android interceptor covers a complex driver/kernel interface with 22 distinct functions. We compared the implementation overhead for the Android case study (Scenario 3) with a previously developed FI specialist for the same SUT and injection scenario, written in the same language. The scenario-specific implementation for GRINDER is smaller than $46\,\%$ of the specialist tool SLOC, indicating that the generalist implementation indeed saves re-implementation efforts. None of the considered Android FI scenarios shared reusable SUT-specific code. While Scenario 3 and the specialist have common injection mechanisms, these were not mutually reusable because of their different interactions with other tool components.

In Scenario 4 the scenario-specific implementation effort accounts for $45.5\,\%$ of the overall code. The reason for this increase lies in performance optimizations that we applied to decrease the run time per test. To avoid the time-consuming transmission of mutated drivers from the host system to the emulator during the execution of each test, we chose to include all mutants in the emulator's data image. This change, however, required a number of extensions to the QEMUAbstraction, e.g., to detect when the system was available for loading the mutant. With a per-test transmission, this detection is implicit: if the system is able to receive TCP data, it is also ready to load modules. Moreover, the failure detection logic became significantly more complex due to longer run times of mutation experiments compared to interface injections and the parallel execution of several tests [16].

V. CONCLUSION

In order to avoid re-implementation efforts for common functionality across different fault injection (FI) tests and SUTs, several *generalist* tools have been proposed. While the proposed tools have been applied for diverse FI scenarios by their authors, the corresponding efforts to adapt these generalists for each application scenario have not been assessed. In particular, testers cannot assess if the reuse of such a generalist would outweigh the implementation effort for a specialized tool from scratch. Moreover, as most generalists are not freely available, testers cannot conduct such assessments themselves.

To address this issue we reported our experience with GRINDER, a generalist FI tool that we make freely available under an open source license. We discussed GRINDER's generic SUT interface and its usage in four different test scenarios for two different SUTs. GRINDER's code reuse across these scenarios ranges between $54.5\,\%$ and $72\,\%$.

REFERENCES

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[2] N. Song, J. Qin, X. Pan, and V. Deng, "Fault injection methodology and tools," in *Proc. ICEOE*, vol. 1, 2011, pp. V1/47–V1/50.

[3] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[4] R. Natella, "Achieving Representative Faultloads in Software Fault Injection," PhD thesis, Università di Napoli Federico II, 2011.

[5] H. Schirmeier, M. Hoffmann, R. Kapitza, and D. Lohmann, "FAIL*: Towards a Versatile Fault-Injection Experiment Framework," *ARCS Workshops*, pp. 1–5, 2012.

[6] J. a. Carreira, H. Madeira, and J. a. G. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, 1998.

[7] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, 1995.

[8] P. Koopman and J. DeVale, "The exception handling effectiveness of POSIX operating systems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 837–848, 2000.

[9] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. DSN*, 2009, pp. 379–388.

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

[11] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: generic object-oriented fault injection tool," in *Proc. DSN*, 2001, pp. 83–88.

[12] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer, "NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors," in *Proc. IPDS*, 2000, pp. 91–100.

[13] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: an environment for examining the propagation of errors in software," in *Proc. ISSTA*, 2002, pp. 81–85.

[14] A. Johansson, N. Suri, and B. Murphy, "On the Impact of Injection Triggers for OS Robustness Evaluation," in *Proc. ISSRE*, 2007, pp. 127–126.

[15] T. Piper, S. Winter, P. Manns, and N. Suri, "Instrumenting AUTOSAR for dependability assessment: A guidance framework," in *Proc. DSN*, 2012, pp. 1–12.

[16] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No PAIN, No Gain? The utility of PArallel fault INjections," in *Proc. ICSE (to appear)*, 2015.

[17] AUTOSAR development cooperation, *Official AUTOSAR Website*, http://www.autosar.org.

[18] Google Inc., *Android*, http://www.android.com.

[19] D. Simpson, *Windows XP Embedded with Service Pack 1 Reliability*. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms838661(WinEmbedded.5).aspx.

[20] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP Kernel Crash Analysis," in *Proc. LISA*, 2006, pp. 12–22.

[21] AUTOSAR, "Technical Safety Concept Status Report," Tech. Rep., 2010. [Online]. Available: http://www.autosar.org/fileadmin/files/releases/4-0/software-architecture/general/auxiliary/AUTOSAR_TR_SafetyConceptStatusReport.pdf.

[22] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: ten years later," in *Proc. ASPLOS*, 2011, pp. 305–318.