FAIL*: Towards a Versatile Fault-Injection Experiment Framework

Horst Schirmeier[1], Martin Hoffmann[2], Rüdiger Kapitza[3], Daniel Lohmann[2], and Olaf Spinczyk[1]

[1]Department of Computer Science 12, Technische Universität Dortmund,
e-mail: {horst.schirmeier, olaf.spinczyk}@tu-dortmund.de
[2]Department of Computer Science 4, Friedrich-Alexander-Universität Erlangen-Nürnberg,
e-mail: {hoffmann, lohmann}@cs.fau.de
[3]Institute of Operating Systems and Computer Networks, TU Braunschweig, e-mail: kapitza@ibr.cs.tu-bs.de

***Abstract*—Many years of research on dependable, fault-tolerant software systems yielded many tool implementations for vulnerability analysis and experimental validation of resilience measures. We identify two disjoint classes of fault-injection (FI) experiment tools in the field, and argue that both are plagued by inherent deficiencies, such as insufficient target state access, little or no means to switch to another target system, and non-reusable experiment code.**

In this article, we present a novel design approach for a FI infrastructure that aims at combining the strengths of both classes. Our FAIL* experiment framework provides carefully-chosen abstractions simplifying both the implementation of different simulator/hardware target backends and the reuse of experiment code, while retaining the ability for deep target-state access for specialized FI experiments. An exemplary report on first experiences with a prototype implementation based on existing x86 and ARM simulators demonstrates the tool's versatility.

I. Motivation and State of the Art

Recent technology roadmaps [1], [2], [3] suggest that future hardware designs for embedded systems will exhibit an increasing rate of intermittent errors in exchange for a life extension for Moore's Law—in terms of even smaller device sizes, lower energy consumption, decreased per-transistor costs, and more performance. This bears new challenges for software developers, which must incorporate software fault-tolerance measures to compensate for unreliable hardware while still benefitting from these new designs: An application-specific resource-efficiency/dependability tradeoff must be made, only hardening mission-critical parts of the software stack against hardware faults. The remaining components must economize resource consumption and are endorsed to yield wrong results, or fail in other modes.

Fault-injection (FI) experiments and dynamic trace analyses are common means to analyze a complex software-stack's susceptibility to hardware faults, and to assess the effectivity of previously applied software fault-tolerance measures [4]. Repeating analysis/evaluation and software-hardening steps allows system designers to converge to an application-specific tradeoff eligible for their product.

This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. KA 3171/2-1, LO 1719/1-1 and SP 968/5-1.

In this context, often an ad-hoc solution—highly specific to the assessed software, the current target platform, and a particular fault model—is chosen, resulting in non-reusable tools. As an unfortunate side effect, such tools—although non-negligible efforts were spent on them—are rarely published themselves, hindering experiment reproduction and forcing the community to consistently reinvent the wheel.

Over the last decades, this situation was improved by a multitude of dedicated FI experiment tool suites, each targeting different development phases and fault models, based on a zoo of hardware simulators at varying levels of simulation accuracy, or on physical prototype hardware accessed through debugging interfaces [5]. These tools can be partitioned into *generalists* and *specialists*:

The *generalists* claim a certain level of flexibility regarding the target-platform backend. Among the benefits of this approach is that experiments can more easily be reused on a different platform—e.g., for gaining evidence the tested fault-tolerance measure is not platform-specific, or to move from a simulator backend to a real hardware prototype in later development phases. With GOOFI, Aidemark, Skarin et al. presented such a generic FI framework, abstracting away target systems in a plugin-based architecture [6], [7], and additionally providing extensive pre- and post-experiment analysis methods [8]. Fidalgo et al. [9] describe a generic tool addressing FI via the NEXUS on-chip debugger interface. Another example is QINJECT (David et al., [10]), injecting faults into a target backend utilizing the GDB debugger interface. These approaches have the common disadvantage that the chosen interface between experiment engine and target backend heavily limits access to target-system state, and narrows the possibilities for FI—e.g., obstructing the possibility to inject networking-device–specific faults into QEMU in the latter example.

In contrast, the *specialist* tools are highly specific to a single target. An example is FAUMACHINE (Sieh et al., [11]), which provides access to a large part of its x86 simulator's state, and enables various FI methods, including, e.g., hard-disk faults. David et al. modified QEMU in [12], which also allows for deep simulator state access. But despite the advantage of providing access to the backend's full capabilities, this class of tools is characterized by severe maintainability issues: Deep

state-access usually results in deep intrusion into the backend's code-base. The resulting *tight coupling* between simulator and FI code often complicates or even inhibits exchanging the tool's target backend later on; in the case of tools that were forked from an existing hardware simulator, such as QEMU, keeping in sync with the simulator's evolution is often too arduous, soon resulting in an outdated FI platform. From an experimenter's point of view, the most notable side effect is the fact that his/her experiment setups are bound to the chosen specialist/backend couple and completely unportable, e.g., to re-run the same experiment with prototype hardware or another target CPU.

A compromise between generalists and specialists—combining their strengths regarding target-backend flexibility, experiment reuse, and deep target-state access—seems desirable. The contribution of this article therefore is:

- A novel design approach for a fault-injection experiment framework that allows *switching target backends* with little effort (Sec. II),
- a framework API abstracting away target-backend details and thereby fostering *experiment code reuse* (Sec. II),
- and a first *experience report* from our FAIL* tool prototype implementation (Sec. III).

The paper concludes with a discussion and an outlook on future work in Sec. IV and V.

II. FAIL*: DESIGN AND IMPLEMENTATION

Based on the needs emerging in our DANCEOS project—a research endeavor in the context of fault-tolerant embedded operating systems—and the state of the art described in the previous section, we are developing FAIL*[1] aiming at combining the advantages of generalists and specialists while avoiding their drawbacks. In the following, we elaborate on design decisions regarding the tool's architecture and its API, and give some details on our prototype implementation based on existing x86 and ARM simulators.

A. Architecture

Two main ideas stand behind the architecture of FAIL*: A modularization scheme chosen specifically for a flexible interchangeability of target backends and for distributing experiments in a parallel environment, and an experiment API designed with the right choice of abstractions in mind for experiment portability and implementation ease-of-use.

Fig. 1 gives an overview of FAIL*'s architecture. User-defined experiments (green) are split up by the user in a *Campaign* and a *Fault Injection* part, which communicate by means of *parameter sets*: A FI campaign typically consists of a potentially large amount of independent single experiments that only differ in the specific fault vector, which can be described in a parameter set. At this point, it should be noted that FAIL* is not limited to fault injection. Other possible parameter sets can be series of input vectors of software components allowing extensive integration testing.

[1]Fault Injection Leveraged; the wildcard operator * stands for exchangeable target backends, e.g., FailBochs representing an instantiation with the Bochs x86 simulator as the backend.

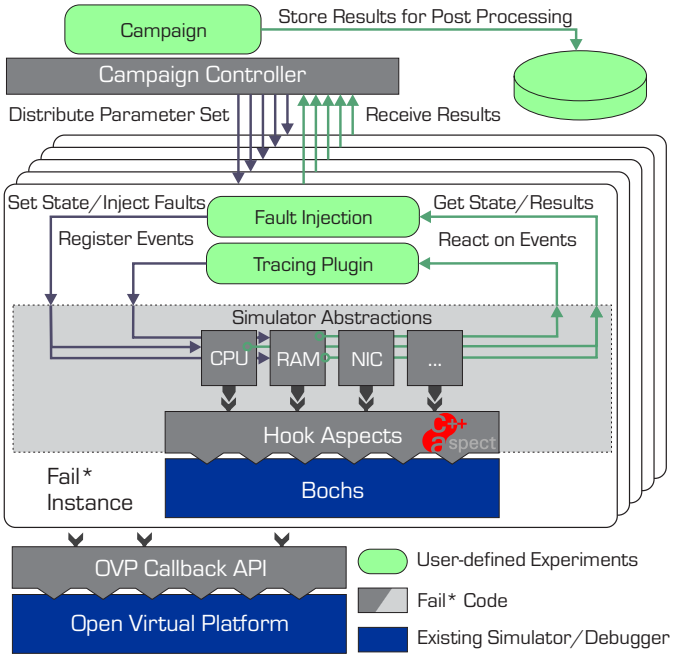


Figure 1. Architecture overview: The *Campaign Controller* distributes parameter sets from a user-defined *Campaign* throughout the FAIL* instances. Each single experiment ("Fault Injection") consumes a parameter set, and controls its target backend through a *Simulator Abstraction* layer. Actual target backends (simulators, but also real prototype hardware) can be exchanged by providing an interfacing module to this abstraction.

The campaign generates a series of parameter sets that are distributed among several (possibly distributed) FAIL* client instances, each iteratively running FI experiments, consuming parameter sets, and communicating back results to the *Campaign Controller*. The FI experiment controls its local target backend through a *Simulator Abstraction* layer, and can be assisted by, e.g., a memory access tracing plugin. Actual target backends (system simulators, or real prototype hardware in later development phases) can be exchanged by providing an interfacing module to this abstraction. The diagram shows a FAIL* instance interfacing with the popular x86 simulator Bochs [13] by means of Aspect-Oriented Programming, a technique we apply to retain a maintainable, loosely-coupled code base while still being able to gain deep access to the simulator's state; though, the details of this method are out of this article's scope and have partially been outlined in earlier work [14].

B. API Design

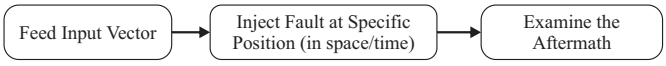
User-defined campaigns and FI experiments are implemented against a C++ API offering access to both target backend meta-information and current state. The interface is designed to abstract away machine-specific details such as the register set or events occurring during an experiment run. The FAIL* API currently provides abstractions for:

- **Machine registers:** Both meta information (e.g., number of registers, platform-independent naming of the program counter or stack pointer registers, bit widths and byte order, an iterator interface) and read/write state access is provided.

- **Memory:** Access to meta data (size, memory type) and state (read/write) is provided.
- **Events:** A set of system events an experiment or plugin may register for, e.g., a specific program address is reached, memory is being written to, or a trap has been generated.
- The **target system** as a whole: Mostly state access is provided, including backend state save/restore for deterministic repeatability of experiments, and a means to reset the system.

Each target backend may additionally introduce interfaces to target-specific state, e.g., a means to manipulate a network device; experiments utilizing these are naturally not portable anymore, unless an adequate abstraction is added to the generic API.

FI experiments usually follow a simple, sequential scheme:



As experiments are event-driven (e.g., wait for reaching the specified position in space/time to inject the fault) but need to retain a substantial amount of internal state between the sequential steps, we chose not to provide a register/callback API (that would force the experiment developers to explicitly carry state from one callback to the next) but an API with blocking calls that return to a sequentially written experiment flow upon event activation (see Sec. III for an example).

Additionally, we currently consider to introduce a classic register/callback API for experiments or companion plugins with little or no state to keep between incoming events; the aforementioned memory-access tracing plugin seems to be such a case and could possibly be formulated even more concisely with callbacks.

C. Tool Prototype

The current prototype implementation of FAIL* provides a target backend for the BOCHS x86 simulator (version 2.4.6) accompanied with all previously described backend abstractions (implemented in C++ and AspectC++ [15], an Aspect-Oriented Programming extension to C++), with an alternative ARM backend (OVP, [16]) currently under development. The campaign parameter set distribution utilizes the Google Protocol Buffer (PB) library for lightweight communication and efficient parallelization. Analogously all result sets are represented as PB messages simplifying post-processing, with the help of PB's versatile language support. As a proof-of-concept, a companion Memory Access Tracing plugin has been implemented.

III. AN EXAMPLE EXPERIMENT

In the following we describe a straight-forward implementation of a fault-coverage campaign using the FAIL* API. The campaign implementation (Listing 1) uses the machine register abstraction to iterate over all registers, every single bit in each register, and all possible instruction offsets within a C function—the analysis subject—running in the target system. For each point in this parameter space, a parameter set is generated

```
// campaign: iterate over all machine registers
RegisterManager& rm =
                simulator.getRegisterManager();
for (RegisterManager::iterator it = rm.begin();
     it != rm.end(); ++it) {
  Register *reg = &(*it);
  // for all bit positions within this register
  for (bitpos = 0; bitpos < reg->getWidth();
       ++bitpos) {
    // for all instruction offsets in the
    // target function
    for (i_offset = 0; i_offset < COV_NUMINSTR;
         ++i_offset) {
      // parameter set for a single experiment:
      FaultCovParam *p = new FaultCovParam;
      p->msg.set_instr_offset(i_offset);
      p->msg.set_bitpos(bitpos);
      p->msg.set_inject_register(reg->getId());

      // enqueue the parameter set for
      // retrieval by a client:
      campaignmanager.addParam(p);
} } }
```

Listing 1. (Simplified) Fault coverage campaign implementation: The code excerpt shows the parameter set generation.

```
// retrieve parameter set from campaign
jc.getParam(par);
// restore previously saved simulator state:
// we're now at the entry of the analyzed func.
simulator.restore("sav/p_entry.sav");
// breakpoint n instructions (defined in
// parameter set) in the future
BPEvent ev_fi_instr(ANY_ADDR,
                    par.instr_offset());
addEventAndWait(&ev_fi_instr);

// FI: single bit-flip in specified register
Register r = simulator.getRegisterManager().
             getRegister(par.inject_register());
r.setData(r.getData() ^ (1 << par.bitpos()));

// Aftermath: traps, timeout, or normal exit
TrapEvent ev_trap(ANY_TRAP);
addEvent(&ev_trap);
BPEvent ev_timeout(ANY_ADDR, 1000);
addEvent(&ev_timeout);
BPEvent ev_func_end(ADDR_FUNC_END);
addEvent(&ev_timeout);
// wait for function exit, trap or timeout
BaseEvent *ev = waitAny();
// experiment result -> parameter set object
if (ev == &id_func_end) {
  int result = simulator.abi_func_retval();
  par.set_resulttype(LOG_NORMAL);
  par.set_result(result);
} else if (ev == &ev_trap) {
  par.set_resulttype(LOG_TRAP);
} else if (ev == &ev_timeout) {
  par.set_resulttype(LOG_TIMEOUT);
}
// communicate result back to campaign ctrl.
jc.sendResult(par);
```

Listing 2. (Simplified) Fault coverage experiment implementation: The code excerpt shows the FI part, parametrized by the register, the bit to flip, and the code offset for injection.

(lines **12–18**) and communicated to an available FAIL* instance, which executes the experiment shown in Listing 2.

The experiment is implemented in a concise, reusable and portable way, based on the FAIL* API. The first action is to request a parameter set for the current experiment from the Campaign Controller (line **2**), holding the aforementioned parameter sets in a job queue. Then we restore a system snapshot taken at the exact point when the function under evaluation is being entered (line **5**). This ensures each run starts under the exact same conditions.

The next steps involve enqueuing a `BPEvent` (BP abbreviates *breakpoint*) that normally fires when a specific address has been reached. In this case, though, the address is not really "specific": `ev_fi_instr` (line **7**) is configured to fire at the wildcard address `ANY_ADDR`, but not on its first occurrence; the optional second parameter (shared by all event types) introduces an event *count*, letting the event only fire after it occurred *count* times, instead of firing at its first occurrence. In effect, this allows us to count down instructions until the point within the evaluated function we want to inject the register bit-flip fault at (the "Instr" element in the parameter set). The blocking `addEventAndWait()` call (line **8**) combines registering as a listener for this event, and waiting for it to fire.

Once we reach line **11**, the event must have fired, and we go for the fault injection. Through the register abstraction and the parameter set received from the campaign, we grab the register we are supposed to inject the bit-flip into (line **11**) and modify its current state in line **13**.

Having injected the fault, we want to observe the outcome of this run: failure-free returning from the function (with a correct or faulty return value—this will be determined offline by evaluating the log files), a hardware trap (MMU violation, division by zero, ...), or a timeout. Lines **16–21** register three more events for catching these cases (without already resuming the simulator: `addEvent()` is non-blocking). The remaining code waits for one of the three events to fire (line **23**: `waitAny()` continues the simulator execution and blocks), and accordingly reports the result back to the campaign controller (lines **25–33**) by storing it in the parameter set (which subsequently is being transmitted in line **34**). Note the abstraction for a target system's ABI convention to store return values (line **26**).

This consequent usage of target backend abstractions allows to carry out the experiment with another simulator or hardware backend, once an abstraction library has been provided for it. Companion plugins, such as the memory-access tracing plugin, allow for a more coarse-grained reuse.

IV. Discussion

We believe FAIL* will achieve the claimed low-effort switching of target backends by its explicit modular design, separating campaign descriptions, experiment instances and the associated target backends. The aforementioned Aspect-Oriented Programming techniques will—at least in the case of the Bochs variant—alleviate the task of updating to newer backend versions: they allow us to reuse traditionally very tightly-coupled modules, such as, e.g., the implementation of FI in the data bus for memory reads. Other systems, such as for example OVP, might already provide distinct callback interfaces which can be utilized directly.

The experiment API (as outlined in Subsec. II-B) and its underlying abstractions for target backend commonalities such as machine registers, memory, or system events was explicitly designed to foster experiment code reuse. The exemplary FI campaign shown in the previous section (Listings 1 and 2) illustrates this quite clearly: it could be reused with another target backend *without modification*, even if FAIL* would be configured for a platform with a completely different instruction set, a reversed byte-order, or another set of general- and special-purpose registers. We are confident to confirm this educated guess once more target backends are implemented.

V. Conclusions and Future Work

We presented a novel concept for a versatile fault-injection framework, aiming at supporting large-scale dependability evaluation and system analysis campaigns on various target-platform backends. Our FAIL* framework provides abstractions supporting portable experiment implementations, fostering code reuse, and reducing the familiarization efforts for new simulator or hardware backends.

Currently our framework implementation is at a relatively early stage, providing a complete interface layer for the Bochs simulator, with OVP interfacing currently in development. Our next steps include advancing the simulator abstraction API, utilizing the available tracing capabilities for conducting pre-injection analyses similar to [8], and implementing an interface to a real hardware platform to evaluate the flexibility of our infrastructure.

References

[1] S. Y. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[2] M. Duranton, S. Yehia, B. de Sutter, K. de Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, "The HiPEAC vision," Network of Excellence on High Performance and Embedded Architecture and Compilation, Tech. Rep., 2010.

[3] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[4] A. Benso and P. Prinetto, Eds., *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation (Frontiers in Electronic Testing)*, 1st ed. Boston: Springer-Verlag, Oct. 2003.

[5] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, Jul. 2004.

[6] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '01)*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2001, pp. 83–88.

[7] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. Los Alamitos, CA, USA: IEEE Computer Society Press, Jul. 2010, pp. 557–562.

[8] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *Proceedings of the 5th European Dependable Computing Conference (EDCC 2005)*, vol. 3463. Springer-Verlag, Apr. 2005, p. 246.

[9] A. Fidalgo, M. Gericota, G. Alves, and J. Ferreira, "Using NEXUS compliant debuggers for real time fault injection on microprocessors," in *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*. ACM Press, 2006, pp. 214–219.

[10] F. M. David, E. Chan, J. Carlyle, and R. H. Campbell, "Qinject: A virtual-machine based fault injection framework," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, 2008, (Poster Presentation).

[11] M. Sand, S. Potyra, and V. Sieh, "Deterministic high-speed simulation of complex systems including fault-injection," in *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*. IEEE Computer Society Press, Jul. 2009, pp. 211–216.

[12] F. M. David and R. H. Campbell, "Building a self-healing operating system," in *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*. Washington, DC, USA: IEEE Computer Society Press, 2007, pp. 3–10. [Online]. Available: http://portal.acm.org/citation.cfm?id=1302491.1302515

[13] K. P. Lawton, "Bochs: A portable PC emulator for Unix/X," *Linux Journal*, Sep. 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=326350.326357

[14] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "Revisiting fault-injection experiment-platform architectures," in *Proceedings of the 17th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '11)*. Pasadena, USA: IEEE Computer Society Press, Dec. 2011, fast abstract.

[15] O. Spinczyk, D. Lohmann, and M. Urban, "Advances in AOP with AspectC++," in *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05)*, ser. Frontiers in Artificial Intelligence and Applications, H. Fujita and M. Mejri, Eds., no. 129. Tokyo, Japan: IOS Press, Sep. 2005, pp. 33–53.

[16] B. Bailey, "System level virtual prototyping becomes a reality with OVP donation from Imperas," *White Paper*, no. June, 2008.