SailFAIL: Model-Derived Simulation-Assisted ISA-Level Fault-Injection Platforms

Christian Dietrich[1,✉], Malte Bargholz[2], Yannick Loeck[1], Marcel Budoj[2], Luca Nedaskowskij[2], Daniel Lohmann[2]

[1] Technische Universität Hamburg, Germany,
✉ christian.dietrich@tuhh.de,
[2] Leibniz Universität Hannover, Germany

Accepted Version. Final Version: https://doi.org/10.1007/978-3-031-14835-4_14

Abstract. For systematic *fault injection (FI)*, we deterministically re-execute a program, introduce faults, and observe the program outcome to assess its resilience in the presence of transient hardware faults. For this, simulation-assisted ISA-level FI provides a good trade-off between result quality and the required time to execute the FI campaign. However, for each architecture, this requires a specialized ISA simulator with tracing, injection, and error observation capabilities; a dependency that not only increases the bar for the exploration of ISA-level hardening mechanisms, but which can also deviate from the behavior of the actual hardware, especially when an error propagates through the system and triggers semantic edge cases.
With SailFAIL, we propose a model-driven approach to derive FI platforms from Sail models, which formally describe the ISA semantics. Based on two existing (RISC-V, CHERI RISC-V) and one newly introduced (AVR) Sail models, we use the Sail toolchain to derive emulators that we combine with the FAIL* framework into multiple new FI platforms. Furthermore, we extend Sail to automatically introduce bit-wise dynamic register tracing into the emulator, which enables us to harvest bit-wise access information that we use to improve the well-known def-use pruning technique. Thereby, we further reduce the number of necessary injections by up to 19 %.

Keywords: ISA-level fault injection · transient hardware faults · simulation-assisted fault injection.

1 Introduction

Shrinking transistor sizes and lowering operating voltages make transient hardware faults, where bits in a machine's dynamic state randomly flip, not only a challenge for safety-critical systems [7, 5, 18] but, increasingly, also for cloud providers [13]. Functional safety standards (e.g., ISO 26262 [15]) already reflect this and recommend explicit measures to assess (and possibly mitigate) the effects of *single-event upsets (SEUs)* causing transient hardware faults (soft errors) [16] on the functional safety of the system. One possibility to assess a system's resilience is a systematic *fault injection (FI)* campaign [23] of the *program-under-test (PUT)*, where many (or even all)

of the possible faults for a single execution are injected into different program re-runs, while the injection platform classifies the subsequent behavior (e.g., benign fault, *silent-data corruption (SDC)*). Such precise failure classification does not only quantify the program resilience, but also guides the introduction of mitigation techniques [14].

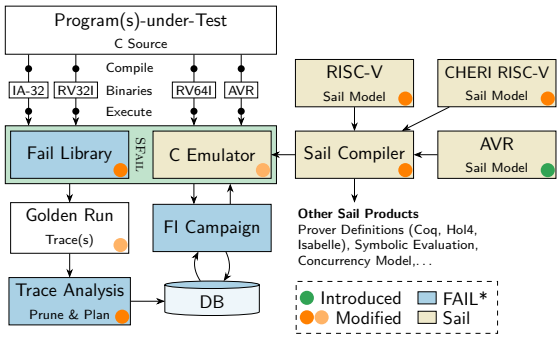


Fig. 1: Overview of SailFAIL

For the injection platform, three solution classes emerged: (1) *Software-Implemented FI* (SWIFI) [24, 3] runs a pre-injected version of the program on the final target platform, leveraging high injection speeds. (2) *Simulation-Assisted FI* (SAFI) [12, 10, 23] uses a low-level architecture simulator (e.g., ISA- or flip-flop level) to inject and observe the executing program "from beneath", which eases parallelization of the campaign. (3) *Hardware-Assisted FI* (HAFI) [6, 9] is similar to SAFI, but loads an instrumented netlist of the target platform loaded into an FPGAs, which allows for gate-level FI at reasonable speeds. While *Software-Implemented FI (SWIFI)*'s pre-injected programs only behave similar (e.g., different code and data layout) to the PUT and HAFI can be parallelized only up to the number of available FPGAs/LUTs, SAFI provides a compromise between campaign run time and result quality.

Still, SAFI requires an instrumented simulator that provides hooks and callbacks for tracing, fault injection, and behavior observation. Often, SAFI platforms extend existing simulators, like Bochs [23] or gem5 [26], which shortens development times but bears the risk that the used simulator differs in subtle details from the actual target platform, which can skew the FI results.

Therefore, with SailFAIL, we propose (see Fig. 1) to derive SAFI platforms from a formal description of the ISA semantic, which allows for easier validation and verification. We chose Sail [1], since Sail models cannot only be translated to theorem prover definitions, but the Sail toolchain can already compile a model to a sequential C emulator. Furthermore, we leverage the existing FAIL* toolchain and combine the generated emulator with FAIL*'s injection and experiment infrastructure. Due to our model-driven approach, we can automatically introduce callbacks during this compilation process, which not only limits the required model modifications but also allows for a fine-grained observation of the PUT down to the bit level. All in all, we are able

to provide five new FAIL* backends for RISC-V (32/64 bits), CHERI RISC-V (32/64 bits), and AVR 8-bit microcontrollers.
The contributions of this paper are as follows:

- We derive ISA-level FI platforms from two existing and one newly developed Sail ISA models. In total, we provide five new FI backends.
- We extend the Sail toolchain to provide for automatically instrumented bit-precise register access tracing.
- We propose bit-wise def-use fault space pruning for partially interpreted register accesses.

The rest of the paper is structured as follows: In Sec. 2, we provide the necessary background on FAIL* and Sail, before we describe the SailFAIL approach in Sec. 3. In Sec. 4, we perform an evaluation of the resulting FI platforms against the existing FAIL* backend for IA-32 and quantify the benefits of our automatic register-tracing transformation, before we conclude in Sec. 5.

2 Background

SailFAIL integrates formal models of the ISA semantic with the existing SAFI framework FAIL* to provide different systematic FI platforms. As our ISA-modeling language, we chose Sail [1], whose toolchain provides versatile backends, including the automatic derivation of ISA-level emulators. Furthermore, different high-quality ISA models (e.g., ARMv8.5, RISC-V, MIPS) with and without the HW-enabled capability extension CHERI [4], which inspired ARM's Morrello CPU extension, are available.

2.1 Systematic Fault Injection

For the systematic FI of transient hardware faults in the volatile machine state, we execute three steps (see Fig. 1, left): (1) *trace* a fault-free program execution as the *golden run*, which spans up the *fault space (FS)* of all potential faults (one fault per time and bit, uniformly distributed). (2) *prune* the FS [25, 11, 20] to plan a representative subset of faults as *pilot injections*. (3) re-execute the program deterministically, *inject* the planned pilots, and classify the following program behavior. While this results in a precise (and even complete) picture of a program's resilience on the chosen level, it is not only time-consuming, but it also requires a specially-instrumented *execution platform* for steps 1 and 3, which must be able to record the program state, inject faults into the executing program, and observe the continued behavior.

For the result interpretation, we have to use metrics that incorporate not only failure counts but also the fault-space size [21]. Otherwise, the results, especially from software-level FI, can deviate substantially from the actual failure behavior [5, 19, 22] of the hardware. However, it was shown [22] that ISA-level injection, if interpreted correctly, is well suited to select the most resilient algorithm variant.

2.2 Sail: ISA Modeling Language

Sail is a special-purpose modeling language for ISA-semantic description, that comes with a toolchain to analyze and translate models to different representations. In its core, the Sail language is a dependently-typed and statically-checked first-order imperative language with strong pattern-matching capabilities that mimics existing industry ISA pseudocode. Sail's main design goal was to create a language that is expressive enough to densely describe ISA semantics but also limited enough to allow for easy translation. For example, Sail does not support higher-order functions.

```
register PC     : bits(22) // Program Counter
register nextPC : bits(22) // PC in the next cycle
register SP     : bits(16) // Stack Pointer

function clause decode 0b1101 @ offset : bits(12)
    = Some(BRTYPE(AVR_RCALL, offset))

function clause execute (BRTYPE(AVR_RCALL, offset)) = {
    // Push return address onto Stack
    let ret = write_dmem(SP,     nextPC[7..0]);
    let ret = write_dmem(SP - 1, nextPC[15..8]);
    SP = SP - 2;

    // Relative jump with a scaling of 2 byte.
    let roffset = signed(offset) * 2;
    nextPC = nextPC + roffset;
}
```

Fig. 2: Sail Fragment of the `rcall` Instruction in our AVR model

In a Sail model, *registers*, which are essentially global variables, contain the model state. Sail supports enums, bit vectors, bit fields, and arbitrary-precision integers as scalar types, while complex types like dynamic lists, vectors, structs and tagged unions are also available. However, most complex and nested types, beyond a simple vector of bitvectors, are quite unusual for typing registers, as the model resembles a hardware implementation. In Fig. 2, we present the relative-call instruction from our own AVR model. `PC` and `nextPC` are 22-bit registers and hold the current and the subsequently following program counter, while `SP` holds the stack pointer and is 16 bits wide.

Sail allows for scattered function definitions that use pattern matching on their argument values. In the example, we show one *clause* of the `decode()` function that decodes `rcall` instructions. When the opcode starts with the pattern `1101`, this clause captures the remaining 12 argument bits in `offset` and returns the decoded instruction. In the corresponding `execute()` clause, we push the return address onto the stack, and perform a PC-relative jump. Sail also supports slicing of bit vectors and has a built-in abstraction for memory, which is accessed through `write_dmem()`.

For generating a sequential C emulator, the Sail toolchain maps registers to global variables and collects all clauses for a function before translating it to a C function. Scalar types and bitvectors, if smaller than 128 bits, are mapped to C integers; bitfields and other complex type become specialized structs with generated accessor functions.

2.3 FAIL*: Fault Injection Leveraged

FAIL* [23] is a versatile FI platform for the injection of transient hardware faults on the ISA level that is designed to support multiple, simulator- and hardware-debugger–based, injection backends (see Fig. 1). At the moment, FAIL* already has support for Bochs (IA-32), Gem5 (ARM), and OpenOCD (ARM HW) and some experimental and less mature backends (Qemu, Trace32 Tri-Core simulator). Furthermore, FAIL* not only handles the FI itself, but also provides the necessary tooling to record the golden-run trace, plan injections, and to run and distribute the FI campaign onto multiple workers.

For providing high-speed tracing and FI, FAIL* directly links its *client* library into the simulator binary: FAIL* creates a second *co-routine* within the simulator process, which alternates its execution with the simulator's control-flow thread. On specific events, the (modified) simulator switches to the co-routine, which is able to inspect and manipulate the machine state before handing back control. Furthermore, to speed up injections [2], back ends can also support saving and restoring the machine state.

For its functionality, FAIL* relies on callbacks within the simulator to inform the client library about events, provide access to the machine state, and for (re-)storing the machine state. For the existing backends, this hand-crafted connection between FAIL* and the simulator is often rather brittle, which increases the burden of updating the simulator. FAIL*, for example, still ships with a rather ancient version of Bochs 2.4.6 (2011), while 2.7 (August 2021) is already available.

While FAIL* records memory accesses for the golden run, it uses a different route to plan CPU-register injections: FAIL* disassembles (with LLVM or libcapstone) the program binary, and inspects the list of read and written registers for each traced instruction. While such detailed instruction summaries are consistently available for COTS architectures, a hardware developer interested in resilient ISA design would have to maintain, both, the ISA-extension and the FI platform and keep them in sync. Therefore, dynamic register-access tracing, where the FI platform itself records which registers are read, would be beneficial for exploring alternative designs.

3 The SailFAIL Approach

To speed up FI-platform development, we propose SailFAIL, a methodology to derive FI platforms from Sail models. Thereby, we not only provide more faithful ISA-level execution platforms for SAFI, but become able to modify the Sail compiler to unleash dynamic *register tracing*: The SailFAIL-derived simulators dynamically report register reads and writes, down to individual bit-field members, which allows for *bit-wise* def-use pruning of partially interpreted registers (e.g., status registers). Furthermore, we extend FAIL* to systematically support backends with multiple types of memory throughout the complete FI toolchain.

3.1 Connecting Sail and FAIL*

Although Sail is designed as an ISA modeling language, it has no built-in notion of traps, executed instructions, or the current program counter, and does not distinguish

Table 1: Callbacks that the emulator calls to inform FAIL* about the execution progress.

`willExecute(PC)`	Indicate the next program counter to execute.
`executeRequests()`	Give FAIL* the chance to save and restore the machine state.
`setIF({true, false})`	Indicate that following memory-accesses stem from the instruction fetch.
`onTrap(num)`	A synchronous trap occured.
`onInterrupt(num, nmi)`	An (non-maskable) interrupt occured.
`didExecute(PC, opcode)`	Execution of instruction did finish.

between ISA-defined and model-specific registers. While this leaves more room for model designers, it requires SailFAIL to use manually inserted callbacks to indicate the current progress of the execution. Tab. 1 gives an overview about the necessary state-transition callbacks. However, since these callbacks are directly inserted into the ISA model, they can co-evolve with the model instead of being managed separately, as currently done in FAIL*.

For memory accesses, we also rely on explicitly inserted callbacks (`onMemoryRead()`, `onMemoryWrite()`) to report to FAIL*. While a generic modification of Sail's memory abstraction would have been possible, explicit hooks allow us to gather additional information, like the type of accessed memory (i.e., RAM or flash), that is only available within the model.

For register FI, we must have read/write access to all ISA-visible registers (e.g., general-purpose registers, program counter…). For this, we manually curate a register mapping with one line for each register to connect the global variables in the C emulator with the FAIL* machinery. From these mappings, we automatically derive the register-access functions and generate the machine-state checkpointing functionality.

We also explored the possibility of automatically exporting such mappings from the Sail compiler. However, since Sail does not distinguish between model-internal and ISA-visible registers, we decided to stick with the curated variant for now. This also has the benefit of making it easier to support registers whose ISA-level format does not match the format within the model. For example, the CHERI RISC-V model stores capabilities in a decoded form, whereby we have to en-/decode them on access.

3.2 Systematic Register Access Tracing

While deriving emulators from formal ISA models, which are easier to validate, already increases the faith in the FI platform, we can harvest more benefits from this model-driven approach. Since registers are a core concept of Sail, SailFAIL is able to use a modified Sail compiler that inserts fine-grained register access tracing into the emulator. Thereby, we no longer require a disassembler, but also gain access to dynamic and more fine-grained tracing information: Even if an instruction statically depends on a register, it does not necessarily interpret all bits or read it in a specific context. For example, some instructions interpret/update only some bits of the machine status word and the exception table base register is only of interest in case of a trap.

To provide detailed access information, including a bit mask of potentially interpreted bits, we modified the C backend of the Sail compiler to insert code at every variable access, which covers all (global) register and (local) variable accesses. However, since Sail supports references to registers and nested data types, correlating an access back to a register definition is not straight-forward. To solve this, we search, before each access, for the memory addresses of accessed values in the previously mentioned register mapping (see Sec. 3.1), which also stores pointers to the mapped global variables. On match, SailFAIL reports the dynamically occurred access to FAIL*, whereby we can cover all ISA registers and report only those accesses that actually occurred.

Nevertheless, there are several challenging patterns in Sail's generated emulators: For accesses to nested register (e.g., within a struct), the emulator creates a temporary copy (with a different address), performs the access, and writes back the result. To catch these accesses, we let the emulator create temporary mappings for the life-time of the copy. Furthermore, we precisely track bit field accesses, which the emulator implements in specialized accessor functions that extract and update specific bits: For each accessor, we calculate an access mask, let the accessor identify the accessed register through the register mapping, and report register and access mask to FAIL*. Thereby, SailFAIL is able to provide precise access information for bit fields, which we subsequently use to reduce the number of necessary FIs.

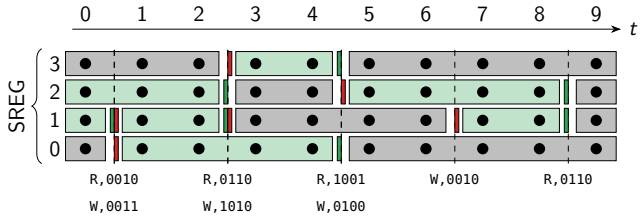
3.3 Bit-Wise Def-Use Pruning

FS size is a major problem for systematic FI that aims for a high, or even complete, coverage of all faults in the FS. Naively, we would have to inject every bit of information in every cycle, which quickly becomes infeasible for realistic programs. Therefore, *fault-pruning methods*, which form sets of *equivalent faults* that all show the same erroneous behavior on injection, are used. For each equivalence set, one pilot injection is performed as a representative.

Of these methods, *def-use-pruning* [25, 11] is the most established one. For a specific fault location, we partition the time axis at read and write events into compact intervals. As faulty information only becomes active on access, every fault within the interval can act as a representative injection, whose failure classification can be projected onto all members of the interval. Even more, intervals that are closed by a write event are surely benign and require no injection. For example, Fig. 3a shows a 4-bit, 10 cycle FS for a processor's status register `SREG` (40 faults). As there are 4 read events (after cycle 0, 2, 4, 8), def-use pruning would schedule 16 injections for complete FS coverage. In a recent work [20], we also made def-use-pruning aware of the program's data-flow to form two-dimensional equivalence intervals.

However, as touched on before, instructions do not necessarily access or overwrite all bits of a register. For example, the first access to SREG in Fig. 3a, only reads bit 1 and writes bits 0 and 1; bits 2 and 3 remain untouched, whereby no injection for these bits is necessary at cycle 0. For bit 3, no injection is even necessary at all, as it is overwritten after cycle 2. In total, with the recorded access-mask information, only 7 injections are really required to cover the presented FS.

To incorporate the access masks, we extend the byte-granular def-use pruning of FAIL* (see Fig. 3b): For each coarse-grained fault location, we keep a FIFO stack



(a) Example fault space with 4-bit **R**ead and **W**rite masks.

```
def bit_wise_prune(events):
    # Initialize Access Stacks and iterate over all events in
    # chronological order
    access_stacks = {loc: Event(time=0,mask=0xffff)
                     for loc in  all_locs])
    for event in events:
        access_mask  = event.mask # Copy mask!
        # Iterate over previous events in reverse
        access_stack = access_stacks[event.loc]
        for prev in reversed(access_stack):
            # Has previous event touched the same bits
            overlap = access_mask & prev.mask
            if overlap != 0:
                # New equiv. interval: (prev.time, event.time)
                new_interval(prev.time, event.time,
                             event,    overlap)
                # Clear bits in both masks
                prev.mask ^= overlap
                access_mask  ^= overlap
            # Pop Empty events from Stack
            if prev.mask == 0: ...
        # Push current event onto the Stack
        access_stack.push(event)
```

(b) Algorithm in Python Pseudocode

Fig. 3: Bit-Wise Def-Use Pruning. For Sail bit fields, SailFAIL records access bit-masks for each read/write event, whereby a more precise def-use pruning is possible.

that holds previous read/write events with a bit-mask of still *open* equivalence intervals. For each access in the golden run, we search the access history backwards for masks that `overlap` with the current access mask. On match, we report an interval (`new_interval()`) between the previous (`prev.time`) and the current access (`event.time`) with the overlapping bit mask. Since we close those reported intervals in previous accesses, we can drop old events (not shown) such that the stack for a register never grows larger than the register width.

3.4 Virtual Fault Spaces

With the flexibility of SailFAIL's model-driven approach, the FI platform has the chance to support a wide range of processors. However, this also requires unified support for the different state holding elements (i.e., registers, RAM, EEPROM, flash…). Therefore, we introduce the *virtual fault space*, which maps the different kinds of memory into a unified FS abstraction, on which we can carry out FS analyses, the fault pruning and

the campaign management. Thereby, SailFAIL is able to handle different architectures, even with experimental ISA extensions, with the same toolchain and the same database schema.

In essence, the virtual fault space maps different fault locations within the target architecture into a linear address space for which we use 64-bit-wide addresses. While recording the golden run, we translate accesses into this unified FS, let FAIL* work on this representation, and only map the FS address back to the actual emulator register (and its corresponding global variable) at injection time. In this translation step, we are also able to provide a dense encoding for unusual kinds of memory. For example, the CHERI RISC-V 32-bit architecture stores one out-of-band tag bit for every 64 bits of memory to ensure the "unforgeable" attribute of capabilities. With the virtual fault space, we are able to densely store those tag bits in a separate FS region instead of supporting 65-bit memory throughout the whole FI toolchain.

4 Evaluation

We used SailFAIL with three Sail models (RISC-V, CHERI RISC-V, AVR), whereof the two RISC-V models were built by the Sail developers [1] and the AVR model was developed by us. Since the RISC-V models have configurable bit widths (32/64 bits), we provide five new backends for FAIL*. After developing SailFAIL for the RISC-V FI platforms, it took one developer day to derive the AVR FI platform. In the following, we will quantify the efficiency of these backends, report on the coverage of the dynamic register tracing, and show the potential saving of bit-wise pruning for bit-packed CPU registers.

4.1 Simulation Overheads

For systematic SAFI, the simulation platform executes the same program, over and over again, potentially millions of times. Thereby, the run-time-overheads for checkpointing and instruction simulation become critical properties for SailFAIL's applicability. Therefore, we quantify these overheads by comparing the existing IA-32 backend (Bochs), which is FAIL*'s most mature platform, with our backends for RISC-V 32-bit, which comes closest to IA-32, and AVR. Furthermore, we perform golden-run tracing with Spike[3], the reference ISA simulator for RISC-V. We executed all benchmarks on a 48-core (96 HW threads) Intel Xeon Gold 6262 CPU with 2.10 GHz and 373 GiB of RAM, within a Debian 11.0 Docker container, which we made publicly available [8] to ease the reproduction of our results.

For the benchmark, we execute the golden run trace step, which includes starting the simulator, saving a machine state checkpoint, and performing a fault-free execution of the program, while recording *program counter (PC)* and memory accesses. We focus on the tracing step here, since fault-injected executions are all purposefully different from each other. However, since each FI run consists of checkpoint restoration and program execution, we believe that the results are transferable.

[3] https://github.com/riscv-software-src/riscv-isa-sim

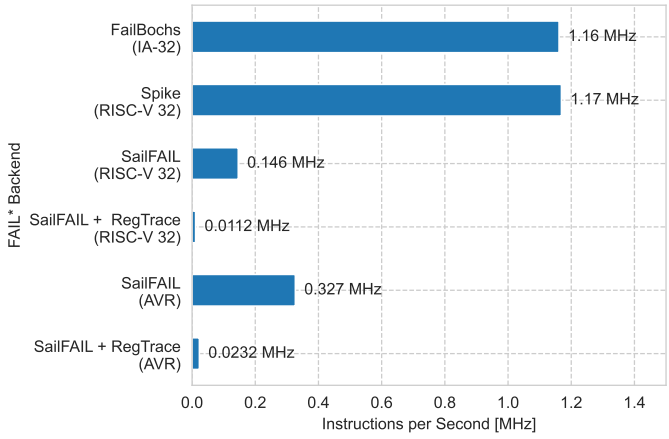


Fig. 4: Simulation Performance

Our PUT calculates the CRC32 checksum over the first 8000 iteratively calculated Fibonacci numbers (overflowed at 32 bits). This program requires almost the same number of instructions on RISC-V and IA-32 (488K instr.). For the single-threaded tracing, we record the number of simulated instructions and the run-time of the executing simulator and show the achieved simulation performance in Fig. 4.

Without dynamic register tracing, our RISC-V 32-bit emulator achieves 13 percent of the performance of Bochs (and Spike), which simulate at around 1.2 MHz. For the considerably simpler 8-bit AVR architecture, SailFAIL achieves 327 KHz. With dynamic register tracing, which is only required for golden run tracing and can be disabled for the actual FI, the simulation frequency is at 11 KHz (RISC-V) and 23 KHz (AVR).

The less-desirable performance results of Sail-generated simulators have two origins: (1) Quality of the model: As the authors admit, the RISC-V model has "many opportunities for optimisation"[17] as it only achieves around 300 KHz without instruction tracing. (2) Sail's C-emulator backend: Although our AVR model is a straight-forward implementation of a rather simple 8-bit ISA, its performance is still by a factor of three from Bochs and Spike. From this, we conclude that the translation from Sail to C is not yet fully optimized. Nevertheless, unlike hand-optimized per-ISA backends, all SailFAIL FI platforms will automatically become faster with improvements to the Sail toolchain.

For checkpoint saving, Bochs requires 0.54 seconds, while the RISC-V emulator only requires 0.024 seconds. These long checkpointing times stem from the fact that Bochs is not only a CPU emulator, but emulates a whole execution platform including periphery. This is also reflected in Boch's startup time of 3.82 seconds, which makes checkpointing absolutely necessary for reasonable FI times. With SailFAIL, the 0.011 second setup time is even faster than saving the checkpoint. So, while the hand-optimized simulator will outperform Sail's generated one in the long run, SailFAIL can already be faster for short running programs.

4.2 Register Trace Coverage

To validate our register tracing approach, we compare the recorded register accesses to the result of FAIL*'s trace analysis, which statically extracts register access patterns from the disassembled binary. At the very least, SailFAIL must record all accesses that can be also be statically extractable. For RISC-V 32-bit, we traced the mentioned program for the first 500 Fibonacci numbers (30 510 instructions), loaded the golden run into the database, and executed (bit-wise) def-use pruning.

In total, the disassembler approach reported $2.58 \cdot 10^5$ register byte accesses[4], while SailFAIL reported $6.27 \cdot 10^5$ accesses. SailFAIL faithfully covered all statically inferred register accesses, but moreover found accesses to six additional architecture-specific registers (`MISA`, `MSTATUS`, `MIP`, `MIE`, `MCYCLE`, `MTIME`), which the CPU implicitly uses to decide on the instruction semantic and interruptions. While these registers are not listed in the disassembler information, SailFAIL makes it possible to also cover them in a FI campaign.

4.3 Efficiency Improvements by Bit-Wise Pruning

Our bit-wise pruning method, combined with the fine-grained access information, allows SailFAIL to cover partially read/written registers without planning a FI for each bit in each accessed register. As the disassembler-based approach would only report full-width accesses to those six registers, the byte-granular def-use pruning method would require $4.43 \cdot 10^6$ pilot injections into the registers. By taking the access masks into account, we are able to reduce this number to $3.58 \cdot 10^6$ ($-19.28\,\%$).

For the CRC32-Fibonacci program on AVR (N=500, 85 511 instructions), the situation looks similar: a disassembler-based approach would require $9.42 \cdot 10^5$ single-bit register injections, while SailFAIL plans $7.9 \cdot 10^5$ injections ($-16.17\,\%$). For AVR, this reduction stems from the 8-bit wide `SREG` register, which, in contrast to RISC-V's machine status word, is essential to the instruction semantic as it stores condition codes. From 21 000 SREG reads, our dynamic tracing recorded that $96.4\,\%$ interpreted a single bit and $3.6\,\%$ accessed two bits. Unlike a byte-wise def-user pruner, which would plan 8 injections per access, our bit-wise pruning only plans one resp. two injections.

4.4 Case Study: SDC Counts for Bubblesort

To demonstrate the flexibility of SailFAIL, we execute a comparative FI campaign to quantify the resilience of different bubblesort implementations on different RISC-V derivatives. With bubblesort, we sort ten integers (register width) that are stored in a *static* array, a *single*-linked list, and a *double*-linked list; for the CHERI variants, the link pointers were capability-protected. Besides RISC-V and CHERI-enabled RISC-V, we also execute our benchmarks on a CHERI-RISC-V variant that we extended with parity-protected capabilities. We also compare the 32-bit and 64-bit ISA variants. We chose these benchmark (variants) as we expect that capabilities, which also provide hardware-enforced bounds checking, positively influence the SDC rate.

[4] FAIL* splits up an access to a 32-bit register in four 4-byte accesses.

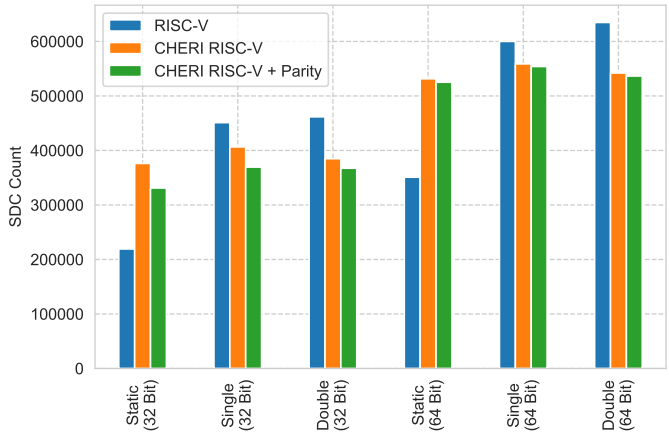


Fig. 5: SDCs for different bubblesort implementations

With our toolchain, we covered the full FS for memory and registers and show the weighted absolute failure counts [21] for the SDC class in Fig. 5. In total, this comparative campaign requires six different FI platforms; each with tooling for tracing, pruning, campaign coordination, and analysis. With SailFAIL, we could provide these toolchains from two basic Sail models with a small modification to the CHERI RISC-V model.

From the results, we can deduce the following observations: (1) If using static arrays, the protection from capabilities does not outweigh the increased attack surface that is induced by managing those capabilities. (2) Our parity extension improves the SDC rate of CHERI RISC-V ISA always and up to 12 %. (3) Although doubling the size of the sorted integers from 32 to 64 bit, the SDC does not increase linearly but between 33 percent (Single Linked List on RISC-V) and 60 percent (Static Array on RISC-V). (4) Using double-linked list instead of single-linked lists the SDC rate decreases for CHERI-protected ISAs, while it increases for the RISC-V without capabilities.

5 Conclusion

With SailFAIL, we derived five new simulation-assisted FI platforms from three formal ISA models written in the Sail modeling language. With limited manual effort, we combined automatically generated C emulators with the FAIL* toolchain, whereby SailFAIL supports all phases of systematic FI campaigns (tracing, injection planing, and injection). We also modified the Sail compiler and let the emulator dynamically record register accesses, down to the level of individual bits. In combination with our bit-wise def-use pruning, we were able to cover implicitly used architectural registers (e.g., machine status words) while reducing necessary injections by up to 19 %. In a case study FI, we compared different (CHERI) RISC-V ISAs and showed that parity-checked capabilities improved the SDC rate by up to 12 %.

References

1. Armstrong, A., *et al.*: ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (2019). DOI: 10.1145/3290384
2. Berrojo, L., Gonzalez, I., Corno, F., Reorda, M., Squillero, G., Entrena, L., and Lopez, C.: New techniques for speeding-up fault-injection campaigns. In: Design, Automation & Test in Europe Conference & Exhibition 2002 (DATE '02), pp. 847–852. IEEE Computer Society Press, Washington, DC, USA (2002). DOI: 10.1109/DATE.2002.998398
3. Carreira, J., Madeira, H., Silva, J.G., and Silva, J.G.: Xception: Software Fault Injection and Monitoring in Processor Functional Units. In: Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA '95), pp. 135–149 (1995). DOI: 10.1145/3290384
4. Chisnall, D., *et al.*: Beyond the PDP-11: Architectural support for a memory-safe C abstract machine. In: Proceedings of the second international conference on Architectual support for programming languages and operating systems. ACM New York, NY, USA (2015). DOI: 10.1145/2694344.2694367
5. Cho, H., Mirkhani, S., Cher, C.-Y., Abraham, J., and Mitra, S.: Quantitative evaluation of soft error injection techniques for robust system design. In: Proceedings of the 50th annual Design Automation Conference, pp. 1–10. ACM New York, NY, USA (2013). DOI: 10.1145/2463209.2488859
6. Civera, P., Macchiarulo, L., Rebaudengo, M., Reorda, M.S., and Violante, M.: An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits. Journal of Electronic Testing 18(3), 261–271 (2002). DOI: 10.1023/A:1015079004512
7. Constantinescu, C.: Trends and challenges in VLSI circuit reliability. Micro, IEEE 23(4), 14–19 (2003). DOI: 10.1109/MM.2003.1225959
8. Dietrich, C., Bargholz, M., Loeck, Y., Budoj, M., Nedaskowskij, L., and Lohmann, D.: SailFail: Model-Derived Simulation-Assisted ISA- Level Fault-Injection Platforms (Software Artifact), (2022). https://doi.org/10.5281/zenodo.6553206. DOI: 10.5281/zenodo.6553206
9. Entrena, L., Garcia-Valderas, M., Fernandez-Cardenal, R., Lindoso, A., Portela, M., and Lopez-Ongil, C.: Soft Error Sensitivity Evaluation of Microprocessors by Multilevel Emulation-Based Fault Injection. IEEE Transactions on Computers 61(3), 313–322 (2012). DOI: 10.1109/TC.2010.262
10. Guan, Q., Debardeleben, N., Blanchard, S., and Fu, S.: F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1245–1254. ACM New York, NY, USA (2014). DOI: 10.1109/IPDPS.2014.128
11. Guthoff, J., and Sieh, V.: Combining software-implemented and simulation-based fault injection into a single fault injection method. In: Proceedings of the 25rd International Symposium on Fault-Tolerant Computing (FTCS-25), pp. 196–206. IEEE Computer Society Press (1995). DOI: 10.1109/FTCS.1995.466978
12. Hari, S.K.S., Adve, S.V., Naeimi, H., and Ramachandran, P.: Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12). ACM Press, New York, NY, USA (2012). DOI: 10.1145/2150976.2150990
13. Hochschild, P.H., Turner, P., Mogul, J.C., Govindaraju, R., Ranganathan, P., Culler, D.E., and Vahdat, A.: Cores that don't count. In: Proceedings of the Workshop on Hot Topics in Operating Systems, pp. 9–16. ACM New York, NY, USA (2021). DOI: 10.1145/2694344.2694367
14. Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., and Schröder-Preikschat, W.: A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes. In: Pro-

ceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering (HASE '14), pp. 33–40. IEEE Computer Society Press (2014). DOI: 10.1109/HASE.2014.14
15. ISO 26262-9: ISO 26262-9:2018: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses. International Organization for Standardization, Geneva, Switzerland (2018). DOI: 10.1145/2694344.2694367
16. Mukherjee, S.: Architecture Design for Soft Errors. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008). DOI: 10.1145/2694344.2694367
17. Mundkur, P., and et.al.: RISCV Sail Model, https://github.com/riscv/sail-riscv – accessed on 2022-02-04. (2015). (Visited on 02/04/2022). DOI: 10.1145/2694344.2694367
18. Nassif, S.R., Mehta, N., and Cao, Y.: A resilience roadmap. In: Design, Automation Test in Europe Conference Exhibition (DATE 2010), pp. 1011–1016. ACM New York, NY, USA (2010). DOI: 10.1109/DATE.2010.5456958
19. Papadimitriou, G., and Gizopoulos, D.: Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers. In: 48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021, pp. 902–915. ACM New York, NY, USA (2021). DOI: 10.1109/ISCA52012.2021.00075
20. Pusz, O., Dietrich, C., and Lohmann, D.: Data-Flow–Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults. In: Proceedings of the 2021 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '21), pp. 97–109. ACM Press, New York, NY, USA (2021). DOI: 10.1145/3461648.3463851
21. Schirmeier, H., Borchert, C., and Spinczyk, O.: Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors. In: Proceedings of the 45th International Conference on Dependable Systems and Networks (DSN '15). IEEE Computer Society Press, Washington, DC, USA (2015). DOI: 10.1109/DSN.2015.44
22. Schirmeier, H., and Breddemann, M.: Quantitative Cross-Layer Evaluation of Transient-Fault Injection Techniques for Algorithm Comparison. In: 15th European Dependable Computing Conference, EDCC 2019, Naples, Italy, September 17-20, 2019, pp. 15–22. ACM New York, NY, USA (2019). DOI: 10.1109/EDCC.2019.00016
23. Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., and Spinczyk, O.: FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In: Sens, P. (ed.) Proceedings of the 11th European Dependable Computing Conference (EDCC '15), pp. 245–255. ACM New York, NY, USA, Paris, France (2015). DOI: 10.1109/EDCC.2015.28
24. Skarin, D., Barbosa, R., and Karlsson, J.: GOOFI-2: A tool for experimental dependability assessment. In: Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN '09), pp. 557–562. IEEE Computer Society Press (2010). DOI: 10.1109/DSN.2010.5544265
25. Smith, D.T., Johnson, B.W., Profeta, J.A., and Bozzolo, D.G.: A method to determine equivalent fault classes for permanent and transient faults. In: Reliability and Maintainability Symposium, 1995. Proceedings., Annual, pp. 418–424. ACM New York, NY, USA (1995). DOI: 10.1109/RAMS.1995.513278
26. Venkatagiri, R., Ahmed, K., Mahmoud, A., Misailovic, S., Marinov, D., Fletcher, C.W., and Adve, S.V.: gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 214–221. ACM New York, NY, USA (2019). DOI: 10.1109/DSN.2019.00033