Data-Flow-Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults

Oskar Pusz
Leibniz Universität Hannover
Germany
pusz@sra.uni-hannover.de

Christian Dietrich
Leibniz Universität Hannover
Germany
dietrich@sra.uni-hannover.de

Daniel Lohmann
Leibniz Universität Hannover
Germany
lohmann@sra.uni-hannover.de

Abstract

In the domain of safety-critical systems, fault injection campaigns on ISA-level have become a widespread approach to systematically assess the resilience of a system with respect to transient hardware faults. However, experimentally injecting all possible faults to achieve full fault-space coverage is infeasible in practice. Hence, pruning techniques, such as def/use pruning are commonly applied to reduce the campaign size by grouping injections that surely provoke the same erroneous behavior.

We describe data-flow pruning, a new data-flow sensitive fault-space pruning method that extends on def/use-pruning by also considering the instructions' semantics when deriving fault-equivalence sets. By tracking the information flow for each bit individually across the respective instructions and considering their fault-masking capability, *data-flow pruning (DFP)* has to plan fewer pilot injections as it derives larger fault-equivalence sets. Like def/use pruning, DFP is precise and complete and it can be used as a direct replacement/alternative in existing software-based fault-injection tools. Our prototypical implementation so far considers local fault equivalence for five types of instructions. In our experimental evaluation, this already reduces the number of necessary injections by up to 18 percent compared to def/use pruning.

CCS Concepts: • **Hardware → Test-pattern generation and fault simulation**; *System-level fault tolerance.*

Keywords: reliability, functional correctness, single event upset, bit flip, fault injection, fault-space pruning

LCTES '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8472-8/21/06…$15.00
https://doi.org/10.1145/3461648.3463851

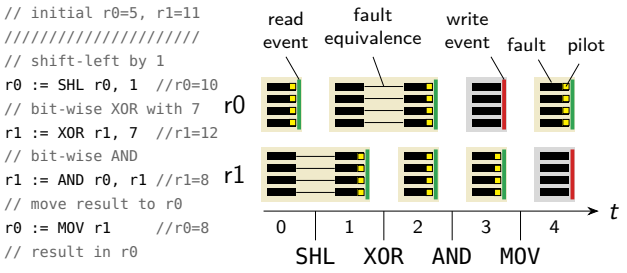
ACM Reference Format:
Oskar Pusz, Christian Dietrich, and Daniel Lohmann. 2021. Data-Flow-Sensitive Fault-Space Pruning for the Injection of Transient Hardware Faults. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3461648.3463851

1 Introduction

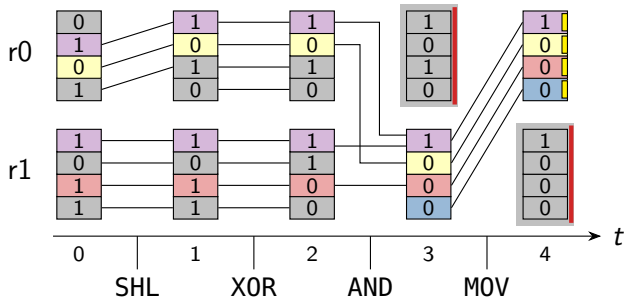
Transient hardware faults have become an emerging challenge for safety-critical systems [11]. Functional safety standards, such as the automotive ISO 26262 standard [26, 27], address with this fact by recommending explicit measures to assess (and possibly mitigate) the effects of *single-event upsets (SEUs)* causing transient hardware faults (soft errors) [33] to the functional safety of the system. This is commonly done by performing extensive *fault injection (FI)* campaigns on the target system [2, 8] that try to mimic either the physical *causes* for SEUs (by exposing the system to, e.g., heat or radiation [17, 38]) or their *effects* (by changing logic signals on, e.g., pin [32], flip-flop [10], ISA [20], or even program level [23]). Our focus for this paper is the injection of logic faults on ISA-level, that is, bit flips in all software-accessible registers and main memory.

To reach full *fault space (FS)* coverage of all possible single-bit faults on this level, one has, in principle, to inject each bit at every cycle of the application's execution – resulting in a prohibitively large number of required experiments. Hence, *fault-space pruning (FSP)* techniques are applied to reduce the number of experiments by grouping injections that lead to equivalent results. The long known *def-use pruning (DUP)* [5, 16, 19, 22, 44] is still considered as a fundamental FSP method, broadly applied by FI tools. Its key advantage is that it is precise and complete, so DUP can be safely applied regardless of the actual *system-under-test (SUT).*

About This Paper. We describe DFPrune, a new data-flow sensitive FSP method that extends on DUP by also using the fault-propagation behavior of instructions when forming sets of equivalent faults. The core idea is depicted in Fig. 1, which presents the example (a) of a code snippet with four instructions (5 machine states) that operate on two 4-bit registers, which results in a FS of 40 injections for full coverage. DUP (b) condenses this FS by constructing



(a) Code Example **(b)** Def-Use-Pruned Fault Space (24 inj.)



(c) Data-Flow-Pruned Fault Space (4 inj.)

Figure 1. Example Program with four Instructions. The machine has two 4-bit registers (`r0`, `r1`) and we expect the result in `r0` after $t = 4$. The yellow boxes indicate the planned pilot injection.

one-dimensional *fault-equivalence intervals (EIs)* spanning between write (*def*) and read (*use*) events, which in this case reduces the number of required injections to 24 (yellow tips). With DFPrune (c), the equivalences span across such *use* events and into different locations by tracking the flow of each bit individually across the respective instructions. Intuitively, this formation of single-bit-fault equivalences is possible until the fault's data-flow may fork into more than one result location, whereby it becomes a multi-bit fault-/error. In Fig. 1c, the flows and resulting equivalence sets are depicted by color: Faults in the gray locations do not influence the program output and are truly benign, while a bit flip in one of the colored bits in step 4 is equivalent to a flip at any other point of the same color. This reduces the number of required pilot injections to 4 (yellow tips).

Like DUP, DFPrune is precise and complete and thereby as generally applicable. In particular, we claim the following contributions:

- The DFPrune method for precise and complete data-flow-sensitive ISA-level fault-space pruning.
- Utilization of value-dependent bit-wise local fault-equivalence rules in ISA-level FSP.
- Quantitative comparison against def-use pruning with campaign-size reductions of up to 18 percent.

The rest of the paper is structured as follows: In Sec. 2, we describe our fault and system model. In Sec. 3, we describe our fault-space pruning method and evaluate its reduction of required fault injections in Sec. 4. We discuss the results of the evaluation and our proposed method in Sec. 5 and the related work in Sec. 6, before we conclude in Sec. 7.

2 Fault and Fault-Injection Model

We want to investigate the resilience of programs in the presence of transient hardware faults (soft errors) [33], which arise from SEUs caused by radiation, electromagnetic interferences or other environmental influences. These influences manifest within the system, in combinatorial logic, registers, and memory cells [7, 25, 43] and surface eventually as bit flips at the hardware/software boundary, where they become visible as ISA-level faults. In the continued program execution, the fault can become benign, cause a detectable failure (i.e., trap, timeout), or propagate as a *silent-data corruption (SDC)* into the calculation results.

In the following, we will define the fault and injection model that we assume for our proposed fault-space pruning technique. We assume that faults happen as uniformly-distributed bit-flips in the volatile program state (memory, general-purpose registers) in between two instructions. To limit the scope of this paper, we do *not* consider faults in the instruction memory, in the instruction semantic itself, or in hardware elements that are not visible on the ISA-level. For a given program execution, this fault model spans the *(complete) fault space (FS)* that consists of one fault for each state bit before (and after) each instruction. In Fig. 1b, each black box represents one fault and the fault space contains 40 faults.

For the program-resilience examination, a FI campaign covers the FS of a single program execution: First, we record a fault-free execution of the program (the golden run) and select a (sub-)set of all faults for injection. For each planned injection, we start the program, execute it deterministically up to the time of the fault, flip the bit in the desired fault location, and continue the program. At this point, we make the assumption that only a single fault happens during one program execution, which is a reasonable assumption at current fault rates [24, 40]. After the injection, the execution platform observes the subsequent program behavior to deduce an *failure classification* for this fault. This classification must be provided by the campaign designer and is specific to the SUT. The FI campaign results in a failure classification for (parts of) the fault space. We can use this fine-grained per-fault classification either to identify vulnerable parts of the program or to summarize the coarse-grained *absolute failure counts* [40].

Besides failure classification, also the *fault-equivalence relation* is important for our proposed method: two faults are *equivalent* if they result in the same deviation from the data-

and control flow and, thereby, result in the same erroneous behavior. If two faults are equivalent they surely fall into the same failure class; but two faults from the same failure class are not necessarily equivalent. Hence, fault equivalence is a strictly stronger relation between faults than failure classification.

As FSs quickly become huge, *fault-space pruning (FSP)* techniques are used to select a representative subset of all possible faults for injection. From these representative *pilot injections*, the pruning method extrapolates the results back onto the fault space. Depending on their strictness and scope, we use the terms *precise* and *complete* to categorize different pruning methods: For individual faults, precise methods predict the same failure classification as if we would perform the actual injection. Complete methods derive a failure classification for each fault-space element instead of only providing a summary of the program resilience. Hence, precise and complete pruning methods are general, safely applicable, and provide us with a fine-grained view of a program's behavior in the presence of transient hardware faults.

3 Data-Flow-Sensitive Fault-Space Pruning

The most prominent example of a precise and complete FSP method is DUP [19, 44], which partitions subsequently following faults in the same location at write (def) and read (use) events into compact EIs. For intervals ending in a read event, a single pilot injection is sufficient as the fault can remains passive until read. Intervals before a write event are marked as *benign* without performing an injection as a fault is surely overridden. In the extrapolation step, DUP projects the injection result back onto all elements of the interval. In Fig. 1b, DUP plans 24 pilots (yellow tips), whereof eight pilots cover two faults, and further eight faults are classified as benign without conducting an injection. The key of DUP's success is its reliance on the strong notion of fault equivalence, whereby it becomes independent of the SUT and the used failure-classification schema.

DUP has two shortcomings when it comes to tracking the propagation of single-bit faults before they escalate to a multi-bit error: (1) DUP only forms one-dimensional EIs along the time axis although instructions (e.g., `MOV r0, r1`) propagate a fault in both dimensions of the fault space. (2) DUP does not consider the operational semantic of the executed instructions although some instructions only forward a fault without spreading it. Hence, we propose *data-flow pruning (DFP)* as a precise and complete FSP method that harnesses this optimization potential while being at least as effective as DUP.

From the golden run, DFPrune constructs a data-flow graph that contains all observed values and executed instructions (Sec. 3.1) and instantiates instruction-local fault equivalences that utilize knowledge about the concrete operand values (Sec. 3.2). From these local equivalences, we calculate

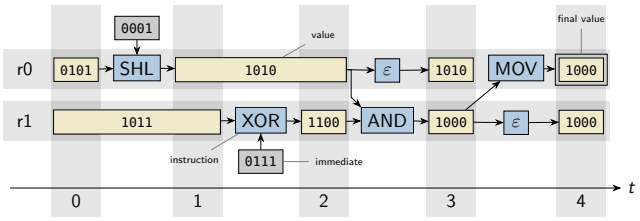


Figure 2. Data-Flow Graph for Running Example from Fig. 1

global *fault-equivalence sets (ESs)*, which can extend across instructions and locations (Sec. 3.3), and select one pilot injection per ES (Sec. 3.4).

3.1 Construction of the Data-Flow Graph

The central data structure for DFP is a *data-flow graph (DFG)* that represents the flow and transformation of data during the golden run. We construct the DFG from the recorded trace, which contains the visited instructions and the observed values. In the following, we will explain its semantics, its construction, and its relation to DUP's FS partitioning into EIs.

The DFG (see Fig. 2) is a directed graph that consists of *instruction nodes* (blue) and *value nodes* (yellow), which occur in a strictly alternating fashion. Value nodes are associated with a specific fault location (e.g., r1), contain a concrete numeric value (e.g., `1011`), and have a temporal extent (e.g., t=[0,1]). An instruction node has an operation type (e.g., `XOR`), uses its predecessor nodes as source operands, while its successors are its calculation result. For completeness, we also model immediate values (gray), which are encoded in the instruction, even though they are not included in our fault model.

In Lst. 1, we show the (simplified) pseudo code for the DFG construction: While we iterate over each instruction of the recorded golden-run trace, the `state` variable keeps track of the current machine state by associating a fault location (e.g., register `r0` or memory address `0x1000`) with its current value, and `time` keeps track of the current position within the trace. Like DUP, we use information about the used fault locations (`src_operands`) and the defined fault locations (`dst_operands`).

First, we collect the source operands: For each used value, we introduce a pseudo instruction (ε) that splits up the source-value node into two nodes and reflects the influence of read events on the fault propagation as a post-read injection only affects the following readers, while a pre-read injection also affects the current instruction. We use the pre-read node as argument for the instruction node (l. 15), but update the machine state with the post-read value and the instruction's result. Whenever a value enters or leaves the machine state, we update its temporal extent (lines 8, 13, 18, 20). For completeness, we also end all value nodes after we have processed the trace (l. 24). Please note that we have not shown

```
def construct(trace, initial_state):
  time  = 0             # time-axis of fault space
  state = initial_state # fault location -> value node
  for instr in trace: # For every golden-run instr.
    # Step 1: read values, create epsilon nodes
    arguments = []
    for operand in instr.src_operands:
      state[operand].end_time = time
      arguments.push(state[operand])

      eps = new Epsilon(state[operand])
      state[operand] = eps.result
      state[operand].start_time = time
    # Step 2: create the instruction node
    op = new (instr.Type)(arguments)
    # Step 3: write back result values
    for operand in instr.dst_operands:
      state[operand].end_time = time
      state[operand] = op.results[operand]
      state[operand].start_time = time
    # Step 4: Forward in time
    time = time + 1
  # Close open equivalence intervals
  for value in state.values():
      value.end_time = time
```

Listing 1. Data-Flow-Graph Construction

value nodes of length zero in Fig. 2 and the intermediate pseudo instructions, which arise if an instruction directly overwrites one of its source operands. After construction, we mark some value nodes as *final* if they are observable results (see Fig. 2).

It is important to note that the set of DFG's value nodes is isomorph to the set of DUP's EIs: For each location, the previous value node is split at read/write events while their temporal extent is equal to the boundaries to the respective EI. However, with its instruction nodes, the DFG includes more semantic information, as it also captures the data-flow relation between value nodes and the instructions that provoke those data flows.

3.2 Instruction-Local Fault Equivalences

Next, we enrich the DFG with semantic information by deriving *instruction-local fault equivalences* that capture the fault-propagation behavior of a single instruction from its source operands to its destination operands. For this, we now focus on a single instruction node and its surrounding value nodes to derive a bit-wise read mask for each source operand as well as a set of bit-wise local equivalence relations. In this derivation, we use our knowledge about the numerical values and assume that at most one input-bit is faulty. In Sec. 3.3, we check inter-instruction conditions to utilize these local equivalences only if this single-fault assumption holds.

With the *read mask*, we mark source-value bit that can have any influence on the instruction. For example, in Fig. 3, the highest bit in a bit-wise left shift cannot influence the performed calculation or its result. While most instructions will interpret all input bits, the read mask captures bits that surely have no influence for this instruction.

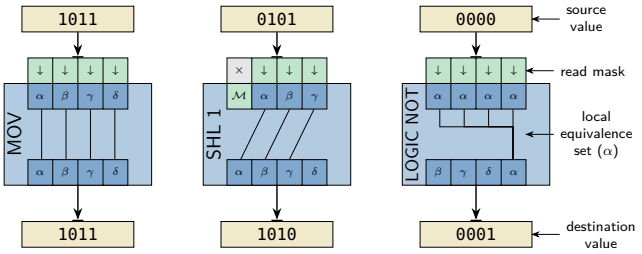


Figure 3. Instruction-Local Fault Equivalences

With the *local equivalence sets (LESs)*, we group input (or output) bits that provoke the same behavior if struck by a bit-flip, whereby it becomes irrelevant into which bit of a LES we inject our assumed single-bit fault. We define LESs to contain an arbitrary number of source bits and at most one output bit. If it contains an output bit, a fault into the any source bit results in flipped output value while no other output bits are affected. If it contains no output bit, each contained source bit will provoke the same faulty instruction result, which can deviate from the golden-run value in multiple bit locations. By creating a single-element LES, we can express that we know nothing about a bit's fault behavior.

Furthermore, we introduce the special value $\mathcal{M}$ to mark input bits that are masked by the instruction such that a fault in the input does not lead to an altered output value. This can either happen if the bit is not interpreted by the instruction (e.g., most-significant bit in the `SHL` input), but also if the semantic of the instruction masks out the fault (e.g., bit-wise `AND` with a 0 in the second operand). Masked bits are not member of any LES.

In Fig. 3, we show the LESs for three instructions where a Greek letters mark LESs. A `MOV` operation propagates a fault in one of its input bits to the same bit position in its output operand. The bit-wise left-shift operation (`shl 1`) moves faulty bits by one position and creates a single-element output LES (δ), whereby we know that the fault δ cannot be provoked by injecting the input values. For the `logic not`, which performs the operation of the `!`-operator in C, a single-bit fault in any of its input bits will result in the same faulty result value (α). Here, we also see that LESs can depend on the numerical operand value: Only if the operand is logically false (`0000`), all bits in the input and the least-significant output bit are member of the same LES.

In our current implementation, we use a small set of manually-derived LES construction rules for five instruction types (`MOV`, `AND`, `OR`, `XOR`, `ADD`) and the ε-instruction, which can be treated as simple `MOV` instruction. For all unknown instructions, we instantiate one single-element LES for every input and for every output bit, whereby the instruction becomes opaque for our fault pruning technique. In Sec. 5, we will discuss how this derivation could be automated by a more formal process.

```
1 def visit(value : ValueNode):
2   # We do not have to inject bits that are not
3   # interpreted as they are surely benign.
4   for bit, count, in read_counts(value):
5     if count == 0:
6       value.symbols[bit] = null_symbol
7
8 def visit(instr : InstructionNode):
9   # Step 1: Back-propagate nulls into read_mask
10  for LES in instr.equivalences:
11    if LES.dst and LES.dst.symbol == null_symbol:
12        for src in LES.sources:
13            instr.read_mask[src.bit] = false
14  # Step 2: Form symbol equivalences
15  for LES in instr.equivalences:
16    symbols = []
17    if LES.dst and LES.dst_symbol != null_symbol:
18        symbols.add(LES.dst.symbol)
19    for src in LES.sources:
20      # Cond. 1: We are the only reader of the value
21      readers = read_counts(src.value)
22      if readers[src.bit] > 0:
23        continue
24      # Cond. 2: The source value must have vanished
25      #    from the system before our result is used.
26      if lifetime(src) > instr.dst.end_time:
27        continue
28      symbols.add(src.symbols[bit])
29    # Mark symbols as equivalent
30    make_equivalent(symbols)
```

Listing 2. Propagation of Global Injection Symbols

3.3 Global Propagation of Fault Equivalences

After the LESs construction, we return our focus on the whole DFG and establish inter-instruction fault equivalences for individual bits. For this, we create injection symbols and propagate them on the DFG under assistance of the LES as long as we can hold up the single-fault assumption. Fig. 4 depicts an DFG (Fig. 1) in different phases of this propagation.

First, we create a unique *injection symbol* for each value-node bit (see Fig. 4a). For example, the symbol "a" denotes a possible injection into the highest-significant bit of `r0` before the `AND` instruction. Furthermore, we introduce a `null_symbol` to mark surely benign injections. In the following, we will try to eradicate symbols by showing their equivalence before planning one *pilot injection* for the remaining symbols.

For this, we perform a data-flow analysis (see Lst. 2) and propagate symbols from latter value nodes back into earlier ones if we can show that both injections are equivalent. As an intuition, we say that two symbols are equivalent if a fault in the earlier symbol will always propagate as a single-bit fault in the latter symbol without having any other influence on the system. In order to form ESs, we perform a backward breadth-first analysis (not shown) from the leaves to the initial state and `visit()` instruction as well as value nodes until no further changes happen.

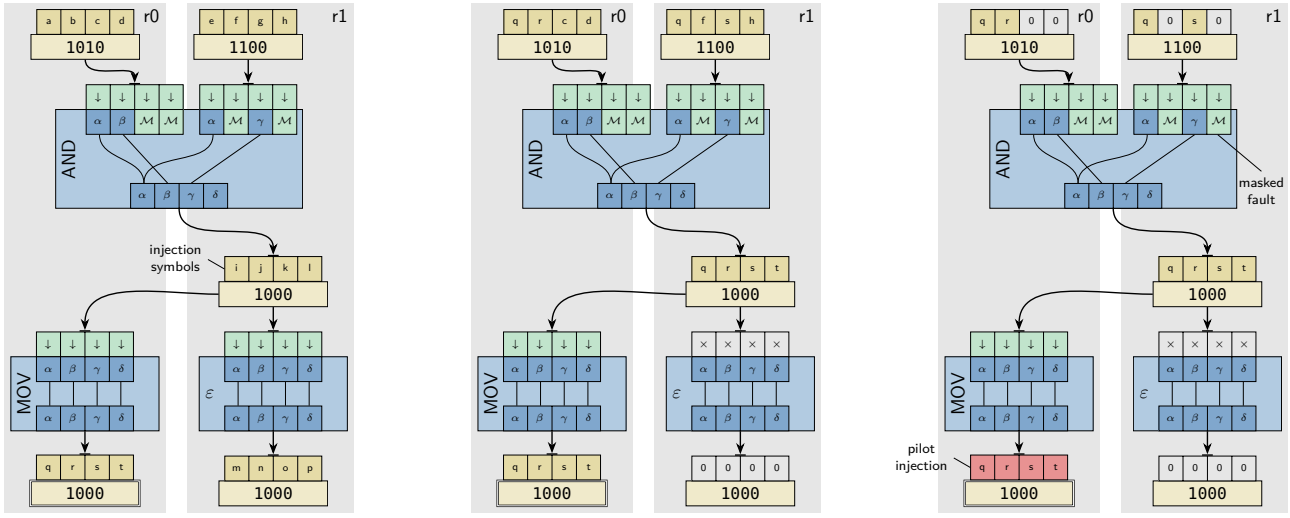
For the propagation, we require two helper functions: For every bit of a value node, the `read_counts()` function calculates the number of instructions that read and interpret it by accumulating their `read_mask` bit vectors. We perform an additional increment, if the value is marked as *final* (see Sec. 3.1), to ensure that bits that influence the observable behavior have at least a read count of one. Furthermore, we require the `lifetime()` helper function that returns the maximal life time of a given value in that fault location. For example, although reading a value from memory ends the EI, the value still remains accessible until it gets overridden. In Sec. 5, we will give a more detailed discussion on the impact of this function and how to improve it. However, for now it is sufficient to know that after the life time of a value ended it must no longer be accessible in the machine state.

When we visit a value node (l. 1), we calculate the read count for every bit and replace the corresponding injection symbol with the null symbol if the count is equal to zero. This is possible since a faulty bit that is not read by any instruction cannot influence the program behavior. In fact, with this propagation, our DFP method catches up with the effectiveness of DUP since *dead values*, which are overridden, have no reader and therefore require no pilot injection. In our running example (see `r1` in Fig. 4b), this propagation rule replaces the injection symbols `m–p` with `0`.

For an instruction node and its LESs, which consist of many source-value bits (`LES.sources`) and one optional destination-value bit (`LES.dst`), we perform two propagations: First (l. 9), for each LES with a destination bit, we propagate null symbols back into the read mask of the current instruction. Since all faults in a LES are equivalent, the fault cannot spread to other destination bits outside the LES and it will end up flipping exactly the destination bit. If this bit has no influence on the result, we can pretend that all source bits are not read by the instruction. By this rule, also all ε-instruction that result in a dead value have a cleared read mask (see `r1` in Fig. 4b).

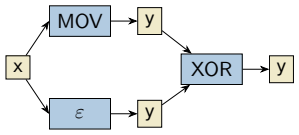
With the second instruction-node rule, we use the LESs to establish fault equivalences between value nodes: we filter the instruction-local LES to include only those bits that cannot spread to other executed instructions but will end up as a single bit-flip at the current instruction's input operands. For a source-value bit to be included into this set it must fulfill two conditions: First, the current instruction must be the only (interpreting) reader of this bit (l. 20). Thereby, we demand that the current instruction is the only instruction in the fault-free execution that interprets this bit.

This first condition prevents us from establishing false equivalences where, for example, a fault can interact with itself (see Fig. 5a): Although, both `MOV` and ε are transparent for a fault in x and move it without spreading to their outputs, we are not allowed to make x and y equivalent, since this



(a) After Construction **(b)** After Propagation **(c)** After Fault-Masking

Figure 4. Data-Flow-Sensitive Fault-Space Pruning for the Running Example from Fig. 1.



(a) Multiple Readers

```
1 void *ptr = NULL;
2 void *tmp = ptr;
3 if (tmp != NULL) {
4   *ptr;
5 }
```

(b) Deviating Control Flow

Figure 5. Problematic Situations for the Fault Propagation

would result in a violation of the single-fault assumption at the XOR node: if the x is flipped from zero to one the XOR-output remains zero, while an injection in y provokes a XOR-output of one.

The second condition (l. 24) is more delicate to understand as it involves dead values that are only read in a deviating execution flows *after* the fault injection. The small (complete) example in Fig. 5b makes the problem more plastic: In the fault-free execution, the dereference instruction (l. 4) is not executed, whereby the DFG contains no respective instruction node that acts as a second reader for `ptr`. However, injections into `ptr` and `tmp` are not equivalent as they can result in different failure classifications: If we inject a fault into `tmp`, the control flow deviates at the condition and the program traps on a null-pointer dereference. But, if we inject a fault into `ptr` such that the pointer coincidentally becomes a valid pointer, the dereference does not lead to a trap and the program terminates successfully. Therefore, we are not allowed to use the assignment's LESs (l. 2) for propagation although it is the only reader in the DFG.

Therefore, we introduce a second condition that ensures that the source value has surely become inaccessible before the propagated fault is used in a following instruction. For this, we check that the life time of the source value is smaller (or equal) to the end time of the destination-value node. Thereby, we ensure that even if the fault results in a deviating control- or data-flow, the fault cannot interact with itself and violate the single-fault assumption. For our implementation, we use the overwrite-time of the value after the last ε-node as a conservative life-time estimation.

However, in some situations, we could use a shorter life-time approximation if we can show that a value already becomes inaccessible earlier: For values that live in registers, the life time ends if no instruction after `LES.dst.end_time` can read the value before it is overwritten. For example, although the value `1000` in `r1` (Fig. 1c) is present until after $t = 4$ it directly becomes inaccessible after $t = 3$ as no instruction in between uses `r1` as an operand.

We mark all source-value bits that meet both conditions and the optional destination-value bit as equivalent and thereby form inter-instruction and inter-location fault-equivalence sets (l. 30). In our running example (see Fig. 4b), we can propagate the symbols `q–t` back, through the MOV and the AND instruction, to the input bits of the AND. This is possible as MOV is the only remaining reader of AND's output, whose life time we could shorten to $t = 3$. Please note that the symbols `{c,d,f,h}` are still present although we will mask them in the next step.

3.4 Fault-Injection Masking and Planning

With the read-mask and the injection-symbol propagation, we can already avoid many pilot injections without sacrificing the precision of the pruning method. However, we have not yet taken the masking capabilities of individual instructions into consideration. For example, a bit-wise AND instruction where both operands are zero will mask out any single-bit fault in its arguments. In the DFG, we already have added per-instruction masking information that is able to

utilize concrete operand values (see $\mathcal{M}$ in Fig. 4b). In the last step, we will use this information to plan only those pilot injections that are not masked by the following instruction.

After our propagation, an injection symbol can occur at multiple value nodes while it indicates equivalent faults. Therefore, for each symbol, we can freely choose one of those value-node-bit-positions and use it as a pilot injection. However, instead of selecting a pilot at random, we choose the pilot with the latest possible injection time as all earlier injection will finally end up as a single-bit flip at this value and the following instruction nodes are the first possibility where the fault could escalate to a multi-bit error. Hence, if all readers of this latest injection symbols mask ($\mathcal{M}$) the input bit, we know for sure that the injection will become benign and we can avoid planning the pilot.

In our running example (see Fig. 4c), for the symbols `q–t`, we choose the latest value node of `r0` and plan four injections. Furthermore, as the `AND` operation masks out several bits, we can avoid the injection of the symbols `c,d,f,h`. As these symbols also occur (see Fig. 1c) in the source and destination operands of the `SHL` and the `XOR` operation, we know that some bits of the initial state are irrelevant for the result.

For the final extrapolation step, we project the injection results back to all value nodes where the symbol occurs. As every value node is an EI in the sense of DUP, we can further project it down to individual faults or weigh the result with temporal extent of the value node. Thereby, DFP is able to deduce an injection result for every element of the fault space, making it a complete pruning method. Furthermore, as all occurrences of a symbol are equivalent, DFPrune's pilot selection and result extrapolation is a precise pruning method.

4 Evaluation

For the evaluation, we apply the DFPrune method to seven benchmarks from the MiBench [18] benchmark suite and to five smaller self-implemented benchmarks, which we call the micro-benchmark suite. We use the DUP technique as a baseline and compare DFP in terms of the required pilot injections and validate DFP's precision with the DUP results. We use our described fault model (see Sec. 2), which covers uniformly-distributed bit flips in registers and memory.

4.1 Experimental Setup

We use the simulation-based open-source FI framework FAIL* [41] for fault injection. This framework extracts program traces, performs DUP, and executes the pilot injections using the IA-32 simulator Bochs. We extended the FAIL* toolchain to support our DFP method.

Due to the client-server-architecture of FAIL* and the independence of the injections, fault-injection campaigns are highly parallelized via FAIL*. We performed the necessary fault injections on a cluster using 17 Intel X5650 @ 2.67 GHz (12 cores each) such that 204 injections can run simultaneously. We performed the pre-injection DUP and DFP as single-threaded programs on an Intel i5-7400 @ 3 GHz.

Table 1. Benchmark Overview

MiBench Benchmarks			
mi/BC	Bitcount	mi/RDD	Rijndael Decryption
mi/BFD	Blowfish Decryption	mi/RDE	Rijndael Encryption
mi/BFE	Blowfish Encryption	mi/SHA	SHA1 Checksum
mi/QSORT	Quicksort of strings		
Micro-Benchmarks			
μ/FIB	Recursive Fibonacci		
μ/LSUM	Iterative sum over an array of integers.		
μ/MIXED	Many bit-wise operations.		
μ/QSort	Recursive quicksort of 3D-Points		
μ/QSortIter	Iterative quicksort of 3D-Points		

4.2 Evaluation Scenario

Our current DFP implementation uses LES rules for the instruction types `MOV`, `ADD`, and bit-wise `AND`, `OR` as well as `XOR`. As described in Sec. 3.2, we treat all the other instructions as opaque such that no equivalence propagation happens across them and each source-value bit results in exactly one pilot injection (like DUP).

We use two groups of benchmarks to evaluate the DFP (see Tab. 1): First, we selected seven programs from the automotive and security branch of the MiBench [18] benchmark suite, as both branches are critical for safe and secure systems. Second, we implemented five benchmarks to cover certain program characteristics: *recursive Fibonacci (μ/FIB)* is dominated by memory-indirect control-flow instructions, *iterative looped sum (μ/LSUM)* has nearly no intermediate results, *mixed bit-wise operations (μ/MIXED)* contains many `ADD`, `AND`, `OR` and `XOR` instructions, and *recursive vector quicksort (μ/QSort)* as well as *iterative vector quicksort (μ/QsortIter)* have data-controlled control flows.

We use six failure classes after the fault injection: (1) Benign (OK) – the program produced the correct result. (2) Silent-data corruption (SDC) – the program terminated but produced an erroneous output. (3) Timeout (TIME) – the program took significantly longer than the fault-free execution. (4) Trap (TRAP) – the processor reported a trap during the execution. (5) Write text segment (TEXT) – the program tried to write to the read-only code section. (6) Write outer space (WRITE) – the program tried to write outside the assigned memory space. With TEXT and WRITE, we emulate the protection capabilities of an memory-management unit.

The fault model (Sec. 2) covers each bit in each of the eight general-purpose registers of an IA-32 processor and each (accessed) main memory bit, while faults happen in between two instructions. We compiled our benchmarks with gcc (version 8.4) at the common optimization level `-O2`. For each benchmark, we recorded a fault-free execution trace, planned pilot injections, and performed all necessary injections to cover the *complete* fault space with DUP and DFP.

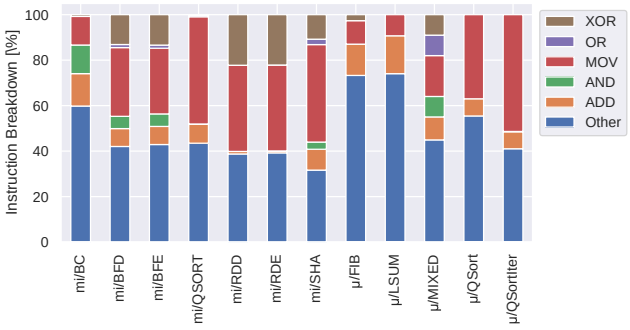


Figure 6. Instruction Statistic of the Golden Run. The instructions of the golden run categorized according to their opcode or other if DFP treats them as opaque.

4.3 Benchmark Details

In the first column group of Tab. 2, we show the number of recorded golden-run instructions for each benchmark. For the MiBench benchmarks, we adjusted the input sizes such that the benchmarks run close to 50 000 executed instructions to keep the campaign size manageable, while providing enough potential for both pruning methods to form large ESs. Furthermore, we show the size of the complete fault space (#Instr. × accessed locations).

Since our pruning method obeys instruction-local equivalences, the distribution of instruction types is a relevant benchmark property (see Fig. 6): Besides categorizing our handled instruction types (`MOV`, `ADD`, `AND`, `OR`, `XOR`), we use *Other* to capture all instructions that DFP treats as opaque. Thereby, we see that DFP considers between 26 percent (μ/LSUM) and 68 percent (*sha1 (mi/SHA)*) for its fault propagation.

`MOV` instructions are the most common with up to 51 percent (μ/QsortIter) and show a great potential for the simplest LES rule of our pruning technique. In two benchmarks (μ/LSUM, μ/QSort), the bit-wise instructions (`AND`, `OR`, `XOR`) are not used at all. Thus, the greatest pruning potential across all benchmarks stems from the operations `MOV` and `XOR`, whereby in certain benchmarks, as in μ/MIXED, the bit-wise instructions make up the greater part of instructions (27 percent) than in others.

4.4 Validation

For validating our pruning approach, we compare the results of DUP and DFP, whereby we show that our method is able to calculate the same complete and precise failure classification for our benchmarks. For this we injected all planned pilots with the same fault-injection setup and record the failure classification in a database. Afterwards, we compare, on a per-fault basis, that both methods yield the exact same classification. In total, DUP covered $1.3 \cdot 10^{10}$ faults with $2.21 \cdot 10^7$ injections. With LES construction rules deactivated, this results (as expected) in the exactly same number of injections. With activated rules, DFP planned $1.93 \cdot 10^7$ injections. In all cases, both methods agreed on the failure classification, while DFP was able to cover the FSs with a lower number of pilot injections.

4.5 Overheads

Calculating the injection pilots via DFP for the micro benchmarks took not longer than 1.6 seconds and for the MiBench ones between 4 minutes (*blowfish decode (mi/BFD)*) and 38 minutes (*rijndael decode (mi/RDD)*). Over all benchmarks, DUP took no longer than 17 seconds (*rijndael encode (mi/RDE)*). However, these relatively long pruning times stem from our current prototypical implementation that visits value nodes more than once. In principle, our proposed method scales linear with the number of instruction and value nodes, since the number of DFG predecessors is limited by the ISA and we have to visit every node exactly once.

Nevertheless, even with our unoptimized DFP, we achieve a significant end-to-end improvements if we compare the pruning time with the campaign's compute-time demand: For mi/RDD, DFPrune executed single threaded for 0.64 hours to reduce the number of injections by 13.13 percent. Virtually executed on a single-threaded machine, the DUP injections would take 188 hours, which we can reduce with DFP by 24 hours, even if we consider our pruning overhead.

4.6 Reduction of Pilot Injections

In the following, we quantify the pilot-count reduction of DFP in comparison with DUP and show the results in Tab. 2: The *fault-space* columns (discussed in Sec. 4.3) characterize the fault space(s). The *def-use pruning* columns contain the number of required pilot injections (#Inj.) and the average weight (⌀Wght.) of a single pilot in the context of the whole campaign. With higher per-pilot weight, a pruning method is able to provide more extrapolated results per performed injection. Under *data-flow pruning*, we show the same information for the DFP with additional columns for the percentual improvements (Δ) the DUP. In the last column group, we further characterize the planned DFP pilots: Among the pilots, *M.Locs.* percent of the DFP pilots cover more than a single fault location, while *M.Nod.* percent cover more than one value node, which is a superset of the pilots refer to *M.Locs.* For DUP, both metrics are zero.

The most important part of Tab. 2 is the percentual reduction of the required FIs (Δ Inj.): For the MiBench benchmarks, we have reductions that range from 10.39 percent (*blowfish encode (mi/BFE)*) up to 18.42 percent (*bitcount (mi/BC)*). With more faults being identified as equivalent, DFP requires fewer injections, which results in an increased weight per injection to guarantee the same complete FS coverage as DUP.

For the micro benchmarks, we see (except μ/LSUM) reductions from 4.36 percent (μ/QSort) up to 14.78 percent (μ/FIB), accompanied by the mandatory weight increases.

Table 2. Comparision of Def-Use Pruning and Data-Flow Pruning. For the given fault space, each pruning method plans #Injections with a given weight. For data-flow pruning, the table also shows the percentage of pilots that cover more than one fault location (M.Locs) and more than one value node/EI (M.Nod.).

	Fault Space		Def-Use Pruning		Data-Flow Pruning				DFP Pilots	
	#Instr.	#Faults [10^6]	#Inj. [10^4]	⌀Wght.	#Inj. [10^4]	Δ Inj. [%]	⌀Wght.	Δ Wght. [%]	M.Locs.[%]	M.Nod. [%]
mi/BC	54334	70.33	222.40	31.63	181.43	-18.42	38.77	+22.58	1.56	3.85
mi/BFD	55495	1894.82	331.38	571.79	295.95	-10.69	640.24	+11.97	2.77	4.92
mi/BFE	54799	1880.82	326.93	575.30	292.97	-10.39	641.98	+11.59	2.73	4.67
mi/QSORT	45225	1623.90	270.58	600.15	234.31	-13.40	693.05	+15.48	3.88	4.62
mi/RDD	70632	3506.17	397.60	881.84	345.37	-13.13	1015.18	+15.12	1.56	4.48
mi/RDE	70316	3457.59	397.99	868.77	351.90	-11.58	982.55	+13.10	1.89	4.53
mi/SHA	40565	242.63	252.79	95.98	219.74	-13.07	110.42	+15.04	5.41	9.02
μ/FIB	2093	1.15	8.87	12.98	7.56	-14.78	15.24	+17.35	2.35	7.51
μ/LSUM	54	0.02	0.26	6.35	0.26	0.00	6.35	+0.00	0.00	0.00
μ/MIXED	89	0.03	0.45	6.61	0.40	-11.83	7.49	+13.42	0.00	6.38
μ/QSort	238	0.18	1.27	14.38	1.22	-4.36	15.04	+4.56	1.84	4.32
μ/QSortIter	777	1.20	4.23	28.43	3.88	-8.18	30.96	+8.90	5.71	7.71

Table 3. Injection Reduction (Δ Inj.) per Failure Class [%]

	OK	SDC	TEXT	TIME	TRAP	WRITE
mi/BC	-1.45	-42.00	0	0	0	-4.69
mi/BFD	0	0	-0.85	-0.32	0	-11.19
mi/BFE	0	0	-0.26	-0.36	0	-10.86
mi/QSORT	-25.96	-2.86	-0.36	-0.15	-0.10	-0.66
mi/RDD	0	0	-1.63	0	-0.07	-13.55
mi/RDE	0	0	-1.13	0	-0.05	-12.12
mi/SHA	0	-22.38	-0.38	-0.04	0	-0.97
μ/FIB	-0.10	-38.72	0	-0.73	-11.77	-15.91
μ/LSUM	0	0	0	0	0	0
μ/MIXED	-27.66	-3.39	-2.04	0	0	-2.12
μ/QSort	0	-5.59	-2.90	-7.41	-5.44	-2.98
μ/QSortIter	0	-15.03	0	0	-2.52	-2.00

For μ/LSUM, DFPrune is not able to reduce the number of injections in comparison to the DUP. However, given the structure of this benchmark, which only accumulates an array of integer values into a register, this is not surprising: The array elements are not overwritten (→ maximal life time) and the program performs no immediate calculation on the accumulator register. Nevertheless, in this case DFP gracefully degrades to DUP.

Next, we characterize the planned DFP pilots: In the last column group, we see that only a small percentage (<5.41% for mi/SHA) of pilots cover more than a single fault location or more than a single value node (<9.02% for mi/SHA). All other pilots represent exact the same set of faults as if they were planned with DUP. This indicates that only a small number of DFP pilots are responsible for our reductions and that if propagation was possible, larger ESs were formed. On average, such multi-value-node pilots span up to 6.9 value nodes (mi/BC).

Next, we consider whether DFP is more likely to combine faults that result in certain failure classifications. For this, Tab. 3 shows how the percentual change in the number of pilots that lead to a certain failure classification. Over all benchmarks, we see that the classes SDC and WRITE often exhibit larger reductions, while the other classes vary widely. Therefore, we conclude that DFP's equivalence propagation is driven more by the fine-grained data flows of the application than the resulting coarse-grained failure classification of the erroneous behavior.

Summarized, compared to DUP, DFPrune results in reductions of up to 18 percent without sacrificing completeness or precision and achieving, even in its unoptimized form, significant end-to-end campaign speed ups.

5 Discussion

In our evaluation scenarios, our prototypical implementation resulted in significantly shorter campaign run times. In the following, we discuss some general benefits and potential disadvantages of our approach, as well as general threats to validity regarding our findings.

Benefits and Disadvantages. The major benefit of DFP is that it is as precise as DUP and relies on fault equivalence, thus, it can be applied as a direct replacement in any FI framework that currently employs DUP to reduce campaign sizes. Hence, the approach is broadly applicable to the domain of ISA-based, precise FI. But also for sampling-based FI, DFP provides benefits as it increases the weight of each injection.

To achieve these benefits, however, one has to implement LES construction rules for the instructions of the respective ISA, which results in a higher one-time per-ISA effort than DUP – and deriving these rules can, in fact, be a tedious and error-prone work if done manually, especially on an

ISA like IA-32. However, as DFP automatically falls back to DUP for unsupported instructions, it is safely possible to do this incrementally, one instruction after the other. Our results with IA-32 show that even a small subset of supported instructions quickly pays off, with `MOV` having the largest impact. On the longer term, however, we intend to generate the rules automatically from a formal ISA specification, such as Sail [3]. In fact, the main reason we did not directly go this route is that for IA-32, which is most supported platform by FAIL*, no reasonably complete Sail model is available.

Even though we consider a campaign-size reduction of up to 18 percent as already useful, there is still much opportunity for further improvement. Firstly, by providing local fault-equivalence rules for further instructions – the currently implemented five instructions cover on average only 51 percent of the executed trace. Secondly, by a more sophisticated life-time analysis of values we should be able to get smaller `lifetime()` values (see Sec. 3.3). In the current implementation, we pessimistically use the overwrite-time as the point a value surely becomes inaccessible. However, as already sketched, it is enough to prove that no control flow, even in the presence of faults, is able to access the value after the inter-instruction fault propagation. For this, it would be necessary to perform a binary analysis to deduce all possible accesses from the point the propagated fault becomes active. This is relatively easy for registers within the same basic block, but becomes more challenging if the value remains present over multiple basic blocks or is located in memory and might require a pointer and alias analysis. Nevertheless, such improvements to `lifetime()` can be provided incrementally with the conservative overwrite-time remaining the default. This is a topic of future work.

Threads to Validity. Our threats to validity are mainly rooted (a) in the construction of the local and global fault equivalences in the DFP as well as in our prototypical implementation and (b) the restricted scope and number of benchmarks on a single ISA.

Regarding (a), we claim precision and completeness of the DFP by providing results that are equal to DUP, which is generally considered to be precise and complete. However, we have not formally proven this equivalence. We think that the straight-forward construction of the DFG (Sec. 3.1) backs our claim by intuition – under the assumption that the local fault-equivalence rules are valid. In fact, if we only apply the LES construction rule for ε-nodes, DFPrune plans the exact same set of pilots as DUP. We furthermore support our claim by our extensive experimental validation, where for all $1.3 \cdot 10^{10}$ faults of our benchmarks, DFP and DUP yield bit-wise equal results. Nevertheless, we consider a formal proof of correctness of DFP as an important topic for future work.

With respect to (b), our selection of benchmarks is a threat to external validity – the benefits may be much lower, if DFP is applied to real applications. We took our benchmarks from the MiBench suite [18] (Sec. 4), as MiBench is an accepted benchmark for covering typical embedded applications. We focused on the automotive and security branches, as we consider them as the best representatives for safety-critical applications. The applied compiler settings (particularly `-O2` optimizations) match the common default of real-world projects. Given that DFP by construction falls back to DUP if it cannot achieve better results, we conclude that by applying DFP one is never "worse off" and that a reduction in campaign size is realistic in general.

As we conducted all our experiments on IA-32, which we chose IA-32 as its implementation is the most complete in the FAIL* framework, our results may not be obtainable on other ISAs. However, we think that DFP would to lead similar or even better results on other ISAs, especially RISC architectures: With our current conservative life-time estimation, our approach works best for values that are kept in registers as they are often use as immediate values that are directly over-written. Tough, IA-32 is a CISC architecture and supports complex memory-addressing schemes, whereby many immediate results do not become visible on the ISA-level.

6 Related Work

Covering a large FS is often infeasible in practice and several methods were proposed to reduce the number of required FIs. The methods can be categorized in terms of *completeness* and *precision* as defined in Sec. 2. Using the precise and complete DUP, which was proposed several times [5, 9, 19, 22, 44], already reduces the number of injections significantly by considering the access times. In addition, DFPrune also uses the instruction semantic to calculate fault equivalences across to form larger ESs.

Similar to our data-flow analysis, but focusing on the whole program instead of a single execution path, Bartsch et.al [6] use *program netlists* [42] to identify faults that surely become benign on all possible program paths. Their method is complementary to DFP, as our approach works for all failure classes but focuses on a single execution path, which will result in a better scalability for long-running programs. Similar to our approach, *SmartInjector* [30] also combines multiple EIs into ESs but only works heuristically, making it an imprecise pruning technique, and they focus on SDCs, while, in many cases, other failure classifications become more relevant [12, 31]. In contrast, DFP is precise and works without knowledge of the failure classifications.

Relyzer [22], and its application to approximate computing *Approxilyzer* [46, 47], analyzes and compares multiple golden-run executions to find faults that might behave equal in all executions. Thereby, similar to our approach, they consider fault masking and propagation at individual operations. However, their work differs in three important aspects from DFP: (1) *Precision*: In its entirety, Relyzer aims for imprecise

pruning, although they in parts rely on DUP to detect equivalences within basic blocks. In contrast, DFP results in a precise fault pruning and thereby could be used as a building block within Relyzer. (2) *Operand Sensitivity*: While their proposed *Constant-based Equivalence* and *Constant-based Masking* techniques also make use of the instruction semantic to prune faults between input and output bits, they only consider operations with constant operands (e.g., `LSHIFT a0, 5`) and ignore the dynamic operand values that can be observed in the golden run. As such operations are relatively rare, they report that these technique only yield minimal improvements. In contrast, DFP considers the constant *and* the dynamic operands of an instruction and thereby is able to provide significant savings over DUP of up to 18 percent. Furthermore, our LES definition is not only able to formulate equivalences between one input and one output bit, but also between several input bits, whereby backwards-directed forking of equivalences becomes possible to express (see `AND` in Fig. 1c). (3) *Pruning preconditions*: We discuss the preconditions (single static reader and inaccessible dead values) on the data-flow to precise inter-location instruction-sensitive pruning on the base of the single-fault assumption (see Fig. 5 in Sec. 3.3). Thereby, we argue that their constant-based equivalence and masking method is inherently imprecise as they do not consider secondary data-flows from the input operands to other instructions.

Nie et. al [34] form ESs by searching similar dynamic instruction sequences and loops of thread executions in GPGPUs, which is complete but not precise. Pusz et. al [35] proposes another heuristic pruning method that only injects data flows that cross basic-block boundaries and extrapolates the result back onto all faults within a basic block, making this approach complete but not precise.

Other sampling methods [28, 36] cover the FS only approximately or concentrate on the most important faults [14]. The *fault-similarity heuristic* [39] uses machine-learning on the recorded machine state to avoid injections of similar faults. Others use structural characteristics, like data-structure dependencies [13], address bounds [37], or memory states [21], to find *similar* faults. However, all of these sampling method are neither complete nor precise.

Besides FI-based resilience assessments, different vulnerability factors [1, 4, 13, 15, 45] combine information about the program execution, the program structure, and the processor architecture to estimate a program's reliability. However, these factors provide no quantitative classification of actual failure behavior and, thereby, provide neither a complete nor precise picture of an actual FS. Trident [29] split a program into blocks, between which they propagate SDC probabilities without performing actual injection. Due to branch probabilities and pruned data dependencies, Trident is neither complete nor precise.

7 Conclusion

With DFPrune, we propose a novel pruning technique for the injection of transient hardware faults that reduces the number of pilot injections that are necessary to achieve complete fault-space coverage. DFPrune performs no heuristic but provides a precise picture of the resulting failure classification as it forms sets of surely equivalent faults by tracking the flow of faulty bits across instructions and locations until a fault could escalate to a multi-bit error. Thereby, DFPrune takes not only the data-access patterns into consideration but also the instruction semantic and the data-flow within the pre-recorded fault-free execution trace.

Compared to the defacto-standard *def-use pruning (DUP)*, we can report reductions of pilot injections for seven programs from the MiBench benchmarks suite that range from 10 to 18 percent while providing bit-wise equal failure classifications. Thereby, our prototypical implementation already provides significantly shorter end-to-end campaign run times while still leaving room for technical and methodical improvements within our provided fault-equivalence-propagation framework.

Try it out!

The source code and the evaluation data can be found here: http://doi.org/10.5281/zenodo.4698901

Acknowledgments

We want to thank Horst Schirmeier for his feedback and our anonymous reviewers for their constructive comments and suggestions. This work has been supported by the German Research Foundation (DFG) under the grant no. LO 1719/4-1.

References

[1] J. Aidemark, P. Folkesson, and J. Karlsson. 2001. Path-based error coverage prediction. In *Proceedings Seventh International On-Line Testing Workshop*. 14–20. https://doi.org/10.1109/OLT.2001.937811

[2] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. 1990. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering* 16, 2 (Feb. 1990), 166–182. https://doi.org/10.1109/32.44380

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3290384 Proc. ACM Program. Lang. 3, POPL, Article 71.

[4] Ghazanfar Asadi and Mehdi Baradaran Tahoori. 2005. An analytical approach for soft error rate estimation in digital circuits. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*. IEEE, 2991–2994. https://doi.org/10.1109/ISCAS.2005.1465256

[5] Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson. 2005. Assembly-Level Pre-injection Analysis for Improving Fault Injection

Efficiency. In *Dependable Computing - EDCC 5*. Springer Berlin Heidelberg, Berlin, Heidelberg, 246–262. https://doi.org/10.1007/11408901_19

[6] Christian Bartsch, Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. 2017. A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems. *Journal of Electronic Testing* 33, 1 (02 2017), 77–92. https://doi.org/10.1007/s10836-017-5643-3

[7] Robert C Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 305–316. https://doi.org/10.1109/TDMR.2005.853449

[8] Alfredo Benso and Paolo Ernesto Prinetto. 2003. *Fault injection techniques and tools for embedded systems reliability evaluation*. Kluwer Academic Publishers, Boston, Dordrecht, London.

[9] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. 2002. New techniques for speeding-up fault-injection campaigns. In *Design, Automation & Test in Europe Conference & Exhibition 2002 (DATE '02)*. IEEE Computer Society Press, Washington, DC, USA, 847–852. https://doi.org/10.1109/DATE.2002.998398

[10] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th annual Design Automation Conference*. 1–10. https://doi.org/10.1145/2463209.2488859

[11] C. Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *Micro, IEEE* 23, 4 (July 2003), 14–19. https://doi.org/10.1109/MM.2003.1225959

[12] Björn Döbel, Horst Schirmeier, and Michael Engel. 2013. Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment. In *Proceedings of the 5th HiPEAC Workshop on Design for Reliability (DFR '13)*. Berlin, Germany.

[13] Mojtaba Ebrahimi, Mohammad Hadi Moshrefpour, Mohammad Saber Golanbari, and Mehdi B Tahoori. 2016. Fault injection acceleration by simultaneous injection of non-interacting faults. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 25. https://doi.org/10.1145/2897937.2898023

[14] Mojtaba Ebrahimi, Nour Sayed, Maryam Rashvand, and Mehdi B Tahoori. 2015. Fault injection acceleration by architectural importance sampling. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE, 212–219. https://doi.org/10.1109/CODESISSS.2015.7331384

[15] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. 2016. ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 168–179. https://doi.org/10.1109/DSN.2016.24

[16] Johannes Grinschgl, Armin Krieg, Christian Steger, Reinhold Weiss, Holger Bock, and Josef Haid. 2012. Efficient fault emulation using automatic pre-injection memory access analysis. In *SOC Conference (SOCC), 2012 IEEE International*. IEEE, 277–282. https://doi.org/10.1109/SOCC.2012.6398361

[17] Ulf Gunneflo, Johan Karlsson, and Jan Torin. 1989. Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*. IEEE Computer Society Press, 340–347. https://doi.org/10.1109/FTCS.1989.105590

[18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 3–14. https://doi.org/10.1109/WWC.2001.990739

[19] Jens Guthoff and Volkmar Sieh. 1995. 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)*. IEEE Computer Society Press, 196–206. https://doi.org/10.1109/FTCS.1995.466978

[20] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. 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. https://doi.org/10.1145/2150976.2150990

[21] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 123–134. https://doi.org/10.1145/2189750.2150990

[22] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. 2013. Relyzer: Application resiliency analyzer for transient faults. *IEEE Micro* 33, 3 (2013), 58–66. https://doi.org/10.1109/MM.2013.30

[23] Martin Hiller, Arshad Jhumka, and Neeraj Suri. 2002. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. ACM, New York, NY, USA, 81–85. https://doi.org/10.1145/566172.566184

[24] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. 1997. Fault Injection Techniques and Tools. *IEEE Computer* 30, 4 (April 1997), 75–82. https://doi.org/10.1109/2.585157

[25] IEC 61508-3. 1998. *IEC 61508-3: - Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. International Electrotechnical Commission, Geneva, Switzerland.

[26] ISO 26262-6. 2018. *ISO 26262-6:2018: Road vehicles – Functional safety – Part 6: Product development at the software level*. International Organization for Standardization, Geneva, Switzerland.

[27] ISO 26262-9. 2018. *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.

[28] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical Fault Injection: Quantified Error and Confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe* (Nice, France) *(DATE '09)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 502–506. https://doi.org/10.1109/DATE.2009.5090716

[29] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 27–38. https://doi.org/10.1109/DSN.2018.00016

[30] Jianli Li and Qingping Tan. 2013. SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis. In *Proceedings of the International Conference on Defect and Fault Tolerance in VLSI and Nanotechnology Systems(DFT '13)*. IEEE Computer Society Press, 236–242. https://doi.org/10.1109/DFT.2013.6653612

[31] Qining Lu, M. Farahani, Jiesheng Wei, A. Thomas, and K. Pattabiraman. 2015. LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults. In *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. 11–16. https://doi.org/10.1109/QRS.2015.13

[32] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. 1994. RIFLE: A general purpose pin-level fault injector. In *Proceedings of the 1st European Dependable Computing Conference (EDCC '94)*, Klaus Echtle, Dieter Hammer, and David Powell (Eds.). Springer-Verlag, 197–216. https://doi.org/10.1007/3-540-58426-9_132

[33] Shubu Mukherjee. 2008. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[34] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault site pruning for practical reliability analysis of GPGPU applications. In *2018*

51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 749–761. https://doi.org/10.1109/MICRO.2018.00066

[35] Oskar Pusz, Daniel Kiechle, Christian Dietrich, and Daniel Lohmann. 2019. Program-Structure–Guided Approximation of Large Fault Spaces. In *2019 24th Pacific Rim International Symposium on Dependable Computing (PRDC'19)*. IEEE Computer Society Press, Washington, DC, USA. https://doi.org/10.1109/PRDC47002.2019.00044

[36] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda. 2008. Statistical Fault Injection. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 122–127. https://doi.org/10.1109/DSN.2008.4630080

[37] Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. 2017. Light-Weight Techniques for Improving the Controllability and Efficiency of ISA-Level Fault Injection Tools. In *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*. IEEE, 68–77. https://doi.org/10.1109/PRDC.2017.18

[38] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. 2017. Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems. In *Proceedings of the 2017 Conference on Architecture of Computing Systems (ARCS '17)*. Springer-Verlag, Heidelberg, Germany. https://doi.org/10.1007/978-3-319-54999-6_1

[39] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2014. Rapid Fault-Space Exploration by Evolutionary Pruning. In *International Conference on Computer Safety, Reliability, and Security*, Andrea Bondavalli and Felicita Di Giandomenico (Eds.). Springer International Publishing, Cham, 17–32.

[40] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. 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, 12 pages. https://doi.org/10.1109/DSN.2015.44

[41] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. 2015. FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)* (Paris, France), Pierre Sens (Ed.). 245–255. https://doi.org/10.1109/EDCC.2015.28

[42] Bernard Schmidt, Carlos Villarraga, Thomas Fehmel, Jörg Bormann, Markus Wedler, Minh Nguyen, Dominik Stoffel, and Wolfgang Kunz. 2013. A New Formal Verification Approach for Hardware-Dependent Embedded System Software. *IPSJ Transactions on System LSI Design Methodology* 6, 0 (2013), 135–145. https://doi.org/10.2197/ipsjtsldm.6.135

[43] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. 2002. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)*. IEEE Computer Society Press, Washington, DC, USA, 389–398. https://doi.org/10.1109/DSN.2002.1028924

[44] D Todd Smith, Barry W Johnson, Joseph A Profeta, and Daniele G Bozzolo. 1995. A method to determine equivalent fault classes for permanent and transient faults. In *Reliability and Maintainability Symposium, 1995. Proceedings., Annual*. IEEE, 418–424. https://doi.org/10.1109/RAMS.1995.513278

[45] V. Sridharan and D. R. Kaeli. 2009. Eliminating microarchitectural dependency from Architectural Vulnerability. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 117–128. https://doi.org/10.1109/HPCA.2009.4798243

[46] R. Venkatagiri, K. Ahmed, A. Mahmoud, S. Misailovic, D. Marinov, C. W. Fletcher, and S. V. Adve. 2019. 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)*. 214–221. https://doi.org/10.1109/DSN.2019.00033

[47] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. 2016. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. https://doi.org/10.1109/MICRO.2016.7783745