Lapse

Automatic, Formal Fault-Tolerant Correctness Proofs for Native Code

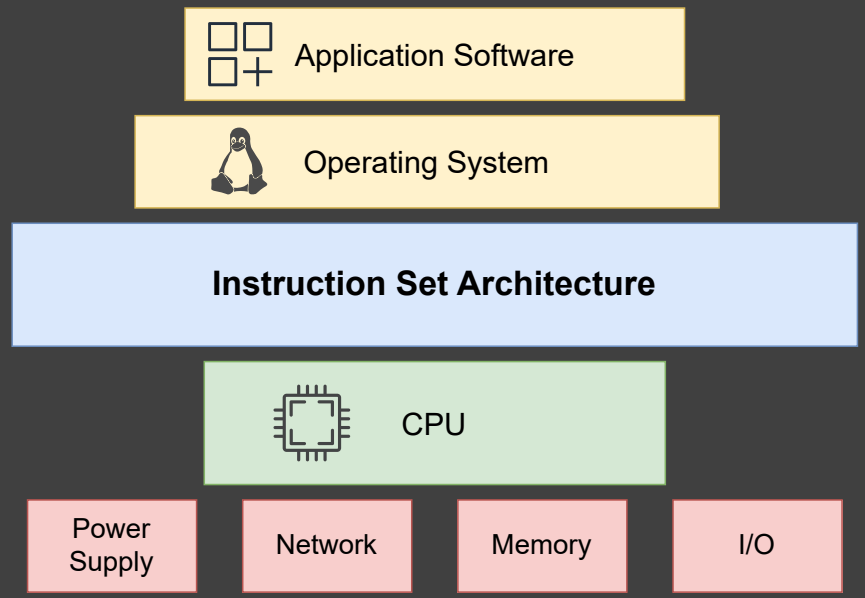
Charles Averill, Ilan Buzzetti, Alex Bellon, Kevin Hamlen

The University of Texas at Dallas
UC San Diego

February 2026

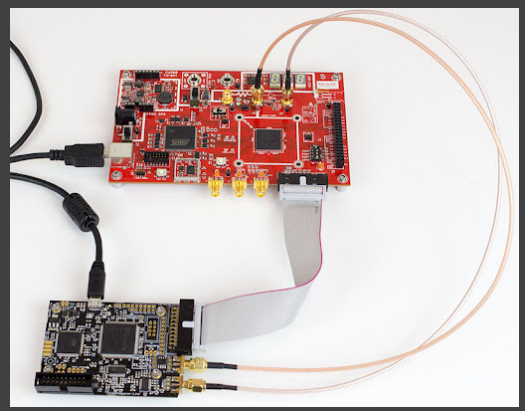
Hardware-Software Contract

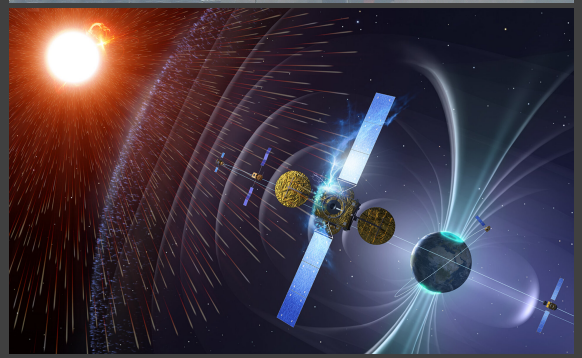
- Users and developers of software trust that hardware behaves as specified in the ISA
- CPU developers design with *hardware invariants* in mind - properties of the physical system that must be constant or within bounds over time
- Some of these invariants are trusted, e.g., "the CPU will receive constant voltage within known tolerances" or "the clock signal's frequency does not drift farther than 15ppm"

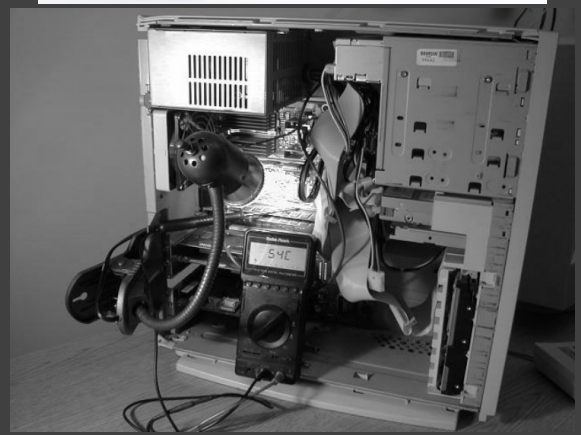


Contract Violations

- Hardware invariants can be invalidated without proper protection, either by harsh or unsuitable environments, or by an adversary
- Many adverse effects: memory corruption, register corruption, **instruction skips**, decoding pipeline corruption, etc.







Software Fault Tolerance

Unfortunately, when developing software for critical systems, we must take extra steps to handle contract violations:

- **Control duplication**: run code multiple times to detect or correct errors
- **Data duplication**: store multiple copies of sensitive data in memory to detect or correct errors
- **Runtime checks** of invariants, state consistency, etc.
- **N-version programming**: implement critical routines in multiple different ways to prevent systematic error propagation
- Fail-stops, state rollbacks, ASLR, ...

How well does any of this work?

Limitations

All of these solutions see practical use in critical systems, but we see some common limits:

- **Complexity**: All of these methods increase the complexity of the target program, which increases the likelihood for implementation bugs
- **Probability**: Many approaches assume that random events (e.g., cosmic ray bit flips) will have random global behavior - but they might not!

Complexity and non-determinism make it very difficult to provide what these methods sought to provide in the first place: **assurance**.

Formal Attempts

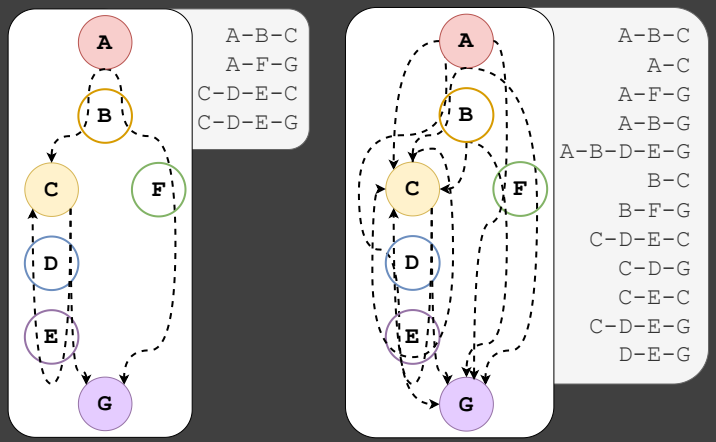
Modern FT efforts often have formal foundations - and limitations!

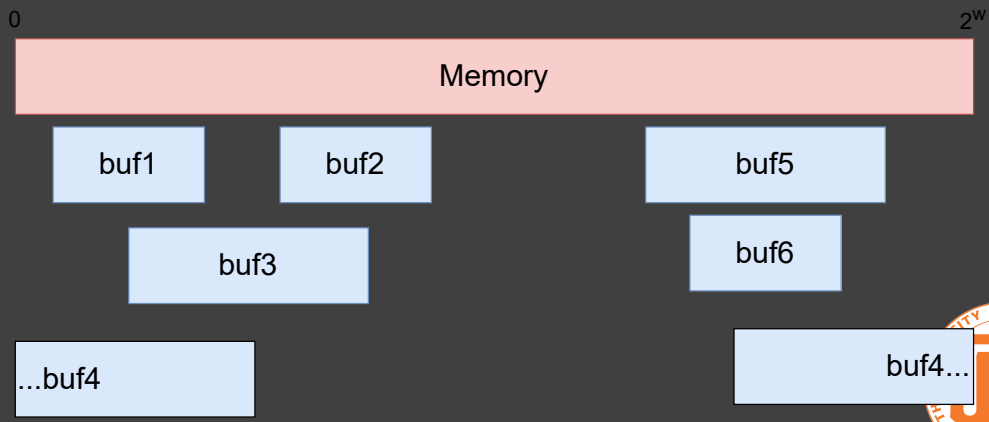
- *PVS*: HOL ITP used extensively at NASA to formally verify fault tolerant systems - but is aimed at program *specifications*, not implementations
- This paper claims to formally verify a binary rewriter that generates skip-tolerant Thumb-2 binaries - but only model checks the rewriter rules, rather than the rewriter tool
- This one claims to verify an AES implementation against fault injection attakcks - but the proof assumes no control-flow attacks and only model checks those scenarios and adds randomization to limit the likelihood

These are steps in the right direction, but reveal that getting this assurance is ridiculously difficult. That's why this problem wasn't solved 50 years ago when bit flips were discovered!

Challenges in Formal Fault Tolerance

- Use of model checking in previous approaches should hint at one big problem: *state space explosion*!
- Common in binary analysis: disassembly is undecidable, CFG recovery is undecidable, binary arithmetic requires expert analysis or heavy SMT solver usage, modeling hardware interaction...
- Framework modeling decisions, expressiveness of host framework, non-determinism, and so on and so on





The Bare Minimum

If you want to get real assurances for real code, you need a system that looks something like:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, and to model faults
- **Flexible intermediate representation** to encode fault behaviors in
- **Machine-checked proofs** at every step
- Bonus points for automation and generality (i.e., multiple ISAs)

A framework for formally verifying all control-flow paths of binary code with baked-in non-determinism, a highly flexible IR in which to encode arbitrary types of *hardware faults* that can be automated and utilized for various architectures. Sounds like a lot of work!

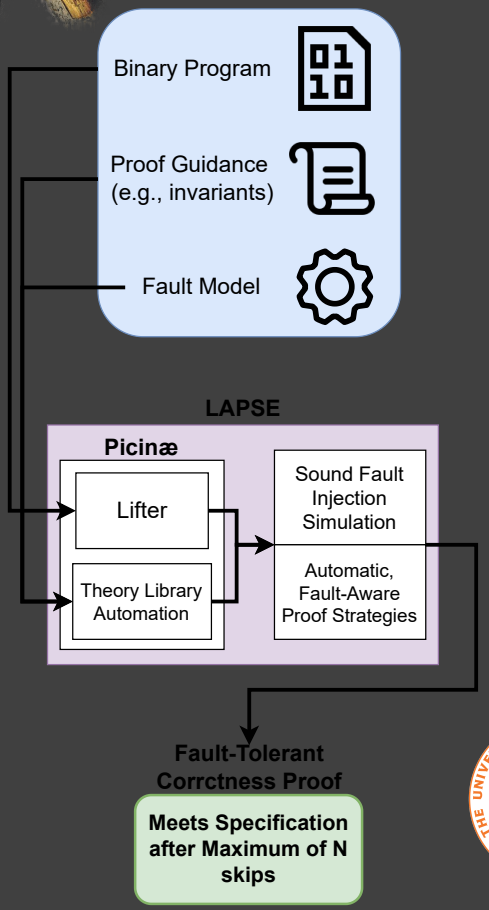
A Lot of Work

Introducing LAPSE[a], the first proof framework for developing machine-checked, fault-tolerant proofs of correctness.

- ISA-generic, instantiated to RISC-V
- Designed with instruction skips in mind, extensible to memory, register, decoder corruption, branch poisoning, ...
- Declarative fault models
- Automation-capable
- Native embedding of non-determinism
- Built in Rocq
- FT proofs follow from FF proofs

[a] "Logic for Analyzing Program Skip Effects"

PICINAE



Lapse Proof Lifecycle

1. Expert analyzes binary, verifies program in fault-free environment
2. User defines `FaultModel` to generate fault tolerance proof machinery
3. Wrap lifted program in `inject_skips` to simulate instruction skips during symbolic execution
4. Write solvers via simple syntactic adjustment of initial proof
5. Launch symbolic execution with solvers to handle invariant sub-proofs

```
Definition inject_skips p s a :=
  match p s a with
  | None => None
  | Some (sz, instr) =>
      Some (sz,
        If (fault_spacing < FT &&
            0 <? FC && Unknown)
        Then
          FC := FC - 1;
          FT := 0
        Else
          FT := FT + 1;
          instr) end.
```

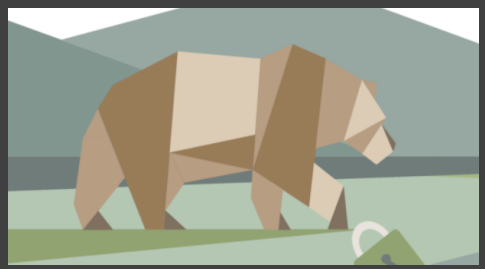
```
Module MyModel <: FaultModel.
  Definition max_faults := 1.

  Definition fault_spacing := 0.
  Theorem fault_spacing_small :
    fault_spacing < 2^32.
  Proof. lia. Qed.
End MyModel.
```

Results

3 example proofs written for to show invulnerability against non-consecutive instruction skip attacks:

- DMR-augmented `CRYPTO_memcmp` from OpenSSL
- DMR-augmented `br_ccopy` from BearSSL
- Triple-Modular Redundant password checker with voting, uses `CRYPTO_memcmp` - non-standard proof structure, interprocedural

OpenSSL
Cryptography and SSL/TLS Toolkit



Ongoing Work

- Implement memory corruption IR transformations, verify spatially-redundant programs
- Survey and simulate fault injection threat models to develop precise descriptions of their effects on software
- Expand evaluation to real-time systems, aerospace applications
- Continue to develop automation primitives for fault tolerance proofs

Team Photo Here

https://charles.systems