

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

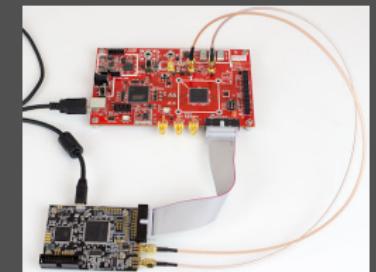
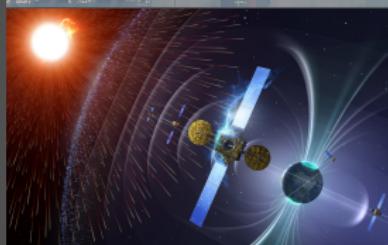


“Using Memory Errors to Attack a Virtual Machine”



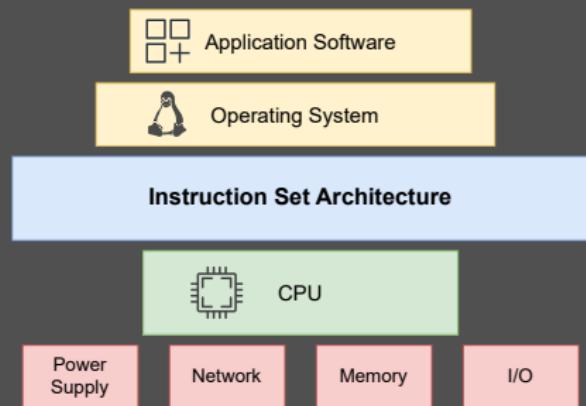
Contract Violations

- 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.



Hardware-Software Contract

- Software developers expect a *contract* between HW/SW
- Contract is “hardware usually tries to behave correctly,” not “hardware always behaves”
- Devs have a responsibility to work within these limits
- CPU developers design with *hardware invariants* in mind, properties of the physical system that must be constant or within bounds over time



Standard Approaches

Software for critical systems must handle contract violations:

- **Control duplication**¹: run code multiple times
- **Data duplication**²: store multiple copies of sensitive data
- **Runtime checks**³ of invariants, state consistency, etc.
- **N-version programming**⁴: multiple unique implementations
- Fail-stops, state rollbacks, ASLR, ...

How well does any of this work?

¹Soft N-Modular Redundancy - Kim, Shanbhag

²Redundancy in Data Structures: Improving Software Fault Tolerance - Taylor, Morgan, and Black

³Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques - Solouki, Angizi, and Violante

⁴N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation - Chen, Avizienis



Standard Approaches

Software for critical systems must handle contract violations:

- **Control duplication**¹: run code multiple times
- **Data duplication**²: store multiple copies of sensitive data
- **Runtime checks**³ of invariants, state consistency, etc.
- **N-version programming**⁴: multiple unique implementations
- Fail-stops, state rollbacks, ASLR, ...

How well does any of this work?

¹Soft N-Modular Redundancy - Kim, Shanbhag

²Redundancy in Data Structures: Improving Software Fault Tolerance - Taylor, Morgan, and Black

³Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques - Solouki, Angizi, and Violante

⁴N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation - Chen, Avizienis



Limitations

Common issues with each approach:

- **Complexity**: more program complexity \propto more bugs
- **Probability**: assume random events (e.g., cosmic ray bit flips) will have random global behavior
- **No Guarantees**: we wanted assurance + safety, we already have FM, none of the approaches give formal guarantees!

How do we get formal assurances for fault-capable environments?



Limitations

Common issues with each approach:

- **Complexity:** more program complexity \propto more bugs
- **Probability:** assume random events (e.g., cosmic ray bit flips) will have random global behavior
- **No Guarantees:** we wanted assurance + safety, we already have FM, none of the approaches give formal guarantees!

How do we get formal assurances for fault-capable environments?



Limitations

Common issues with each approach:

- **Complexity:** more program complexity \propto more bugs
- **Probability:** assume random events (e.g., cosmic ray bit flips) will have random global behavior
- *No Guarantees:* we wanted assurance + safety, we already have FM, none of the approaches give formal guarantees!

How do we get formal assurances for fault-capable environments?



Limitations

Common issues with each approach:

- **Complexity:** more program complexity \propto more bugs
- **Probability:** assume random events (e.g., cosmic ray bit flips) will have random global behavior
- **No Guarantees:** we wanted assurance + safety, we already have FM, none of the approaches give formal guarantees!

How do we get formal assurances for fault-capable environments?



Limitations

Common issues with each approach:

- **Complexity:** more program complexity \propto more bugs
- **Probability:** assume random events (e.g., cosmic ray bit flips) will have random global behavior
- **No Guarantees:** we wanted assurance + safety, we already have FM, none of the approaches give formal guarantees!

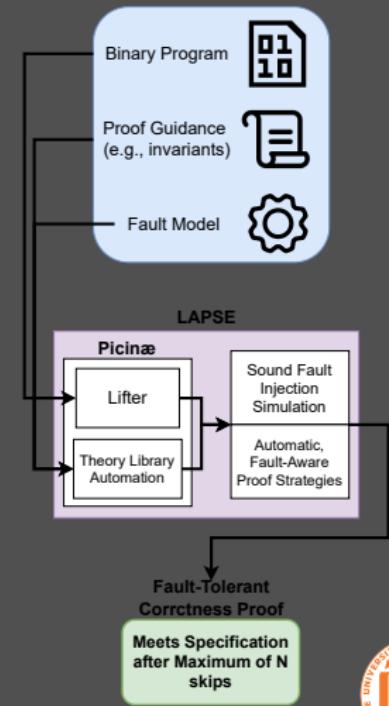
How do we get formal assurances for fault-capable environments?



Formal Assurances for Faulty Environments

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

- Enables construction of machine-checked proofs *assuming a Fault Model*
- Fault model explicitly defines environmental effects
- Get guarantees that code is correct/safe/insert property here within a provided fault model

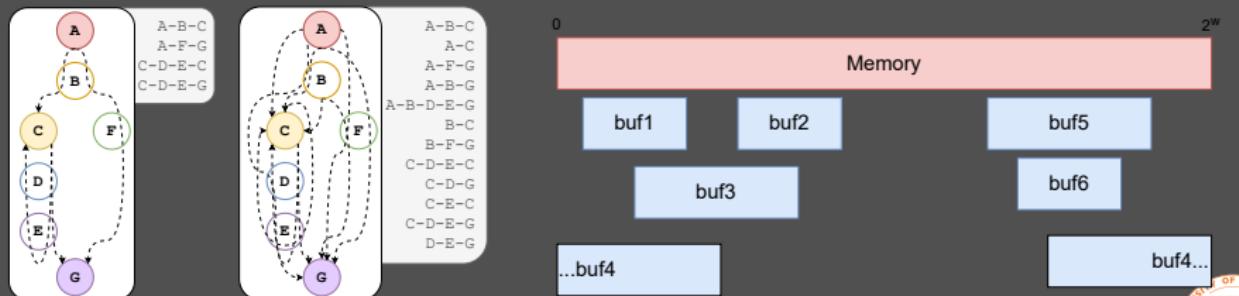


^a "Logic for Analyzing Program Skip Effects"



Challenges in Formal Fault Tolerance

- Common in many FM approaches: *state space explosion!*
- Common in binary analysis: disassembly is undecidable, CFG recovery is undecidable, binary arithmetic needs expert analysis or SMT solvers, modeling hardware interaction...
- Expressing faults, framework modeling decisions, expressiveness of host framework, non-determinism, and so on and so on



Formal Attempts

Modern, formal FT efforts have made great strides in solving this problem:

- **PVS**: NASA HOL ITP used to verify critical systems
- **Moro et al.** formally verify a binary rewriter generating skip-tolerant ARM binaries
- **Patranabis et al.** verify an AES impl against fault injection attacks

Right direction, but shows that this assurance is ridiculously difficult! Still unsolved: state space explosion, sound symbolic execution, scalability, generality...



Formal Attempts

Modern, formal FT efforts have made great strides in solving this problem:

- **PVS**: NASA HOL ITP used to verify critical systems - aimed at program *specifications*, not implementations
- **Moro et al.** formally verify a binary rewriter generating skip-tolerant ARM binaries - but only model checks the rewriter rules, rather than the rewriter tool
- **Patranabis et al.** verify an AES impl against fault injection attacks - but assumes no control-flow attacks, model-checks those scenarios

Right direction, but shows that this assurance is ridiculously difficult! Still unsolved: state space explosion, sound symbolic execution, scalability, generality...



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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!



Technical Challenges

Many challenges to overcome for full proof mechanization:

- **Sound symbolic execution of machine code** to ensure all possibilities are covered
- **ISA semantics with non-determinism** to handle UB, hardware interactions, model faults
- **Flexible intermediate representation** to encode fault behaviors
- **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

- ISA-generic, instantiated to RISC-V
- Designed with instruction skips in mind, but extensible
- Declarative fault models
- Automation-capable
- Native embedding of non-determinism
- Built in Rocq on Picinæ (developed for DARPA V-SPELLS)



PICINAE



LAPSE Proof Lifecycle

1. Expert analyzes binary, verifies program in fault-free environment
2. User defines FaultModel (encodes environmental effects) 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 NonConsecSkips <: FaultModel.
  Definition max_faults := 1.
  Definition fault_spacing := 0.
  Theorem fault_spacing_small :
    fault_spacing < 2^32.
  Proof. lia. Qed.
End NonConsecSkips.
```



Results

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

- DMR CRYPTO_memcmp from OpenSSL — 55LoP, ~1hr, reused
- DMR br_ccopy from BearSSL — 55LoP, ~15mins, reused
- Triple-Modular Redundant password checker with voting, uses CRYPTO_memcmp - non-standard proof structure, interprocedural, 580LoP, ~3 days, ~reused



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



<https://charles.systems>