LAPSE: Automatic, Formal Fault-Tolerant Correctness Proofs for Native Code

Charles Averill
The University of Texas at Dallas
charles@utdallas.edu

Ilan Buzzetti
The University of Texas at Dallas
ilan.buzzetti@utdallas.edu

Alex Bellon
University of California San Diego
abellon@ucsd.edu

Kevin W. Hamlen
The University of Texas at Dallas
hamlen@utdallas.edu

***Abstract*—LAPSE is a new framework for developing fault-tolerant correctness proofs for near-arbitrary native code. It lifts binary code into an intermediate representation (IR) whose operational semantics admit hardware faults. LAPSE implements a machine-verified symbolic execution engine for the resulting IR within the Rocq automated theorem proving framework, creating a proof environment in which the space of possible executions includes all potential fault possibilities. To cope with the increase in proof space, automation tools succinctly describe and reason about the desired fault model. An implementation for 32-bit RISC-V semantics and evaluation on security-critical cryptographic subroutines from OpenSSL and BearSSL demonstrates that fault-aware proofs can be constructed from standard correctness proofs with little additional work, often requiring no novel proof techniques. The results show that developing fault-tolerant correctness proofs is not only feasible, but rote for certain kinds of fault-tolerant programs.**

I. INTRODUCTION

Machine-checked correctness proofs are widely championed as offering exceptionally strong guarantees of safety and security for mission-critical code. For example, formal verification is increasingly utilized in critical systems (e.g., [1], [2]), including open-source libraries and utilities (e.g., [3], [4]), because it yields explicit, complete-coverage proofs of critical code properties that are independently checkable down to the foundations of mathematics. This results in a trusted computing base that can be significantly smaller and more robust than both formal and informal alternatives, such as software model checking or unit testing (cf. [5], [6]).

Like all software verification approaches, formal methods are contingent upon assumptions about the computational environment in which the analyzed code will run. These include the semantics of the *instruction set architecture* (ISA) on which the code is expected to run, such as assuming a single-threaded environment and trustworthy interrupts and context switches (e.g., [7]). Proofs that assume a given ISA semantics offer no assurances about the outcome of running the code on a different ISA, or in an environment where some of the ISA assumptions are unreliable or violated.

Unrealistic assumptions about hardware behavior can therefore engender a false sense of safety, as these assumptions can be invalidated either by a malicious adversary or the physical environment (e.g., extreme heat, cold, or radiation). To exploit this, attackers perform *fault injection attacks*, such as tampering with a processor's power supply or subjecting it to extreme electromagnetic radiation, to cause adverse computational effects. Typical effects include memory corruption, register corruption, modifying instructions in the decoder pipeline, and skipping instructions. Any one of these faults is likely to invalidate the assumptions of the ISA model on which formal proofs rely, rendering the proved result inapplicable.

Realistic fault-aware assumptions are unfortunately difficult or impossible to express in most modern software verification frameworks due to their reliance on source-level code semantics. In particular, most formal methods (a) assume a trusted compiler that preserves the semantics of the source code in the generated machine code (*semantic transparency*), or (b) machine-verify that a particular compiler implementation achieves semantic transparency for a *faultless* ISA (e.g., [3], [8]). Such approaches cannot formulate proofs that reason about faults expressible only at the binary level, including faults in machine instructions, registers, instruction-decoding, and memory, which are abstracted away by most high-level programming languages. Reasoning about program behavior in a fault-susceptible environment thus requires reasoning directly about the program's machine code, in which binary components are a part of the computational model.

Informal alternatives include design patterns for constructing fault-tolerant software (e.g., [9], [10]) and *fault injection testing* [11]. These sacrifice assurance for scalability to achieve a best-effort defense against fault injection attacks in practice. Unfortunately, design patterns can be implemented incorrectly or be undermined by unforeseen compiler behaviors [12]. Fault injection testing involves injecting faults into running software to empirically assess its fault tolerance, which scalably supports many fault models but suffers from the non-generality and low assurance limitations common to unit

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23046
www.ndss-symposium.org

testing approaches.

To address these limitations and facilitate formal analysis of program behavior in fault-susceptible environments, we present LAPSE (Logic for Analyzing Program Skip Effects), a framework for formal verification of native code experiencing arbitrarily many (non-deterministic) hardware faults. LAPSE is built atop the Picinæ [13] formal binary analysis framework, which provides ISA-generic facilities for verified lifting [14] into analysis-amenable IR, and verified symbolic execution for sound traversal of all potential program control-flows. LAPSE's simulation and automation capabilities enable automatic conversion of standard Picinæ functional correctness proofs into proofs that consider environments susceptible to faults. This generates fault-tolerant guarantees of correctness for native code that is exposed to harsh or adversarial environments.

Specifically, we present the following contributions:

1) We present LAPSE, the first framework that affords machine-checked proofs of correctness for production-level binary code in the presence of hardware faults, such as single-instruction skips.
2) We provide machinery for instantiating LAPSE for any ISA, and implement the framework for 32-bit RISC-V.
3) We provide proof tactics for automating fault-aware correctness proofs
4) We prove that augmented implementations of two critical cryptographic functions from OpenSSL and BearSSL are correct even with at most one skipped instruction.
5) We prove that an implementation of a Triple-Modular-Redundant password checker is correct even with at most one skipped instruction.
6) LAPSE is available within the Picinæ repository.

The remainder of this paper is structured as follows. Section II introduces the formal foundations required for our work, including Rocq, the Picinæ binary analysis framework, and existing approaches to fault injection attacks and fault tolerance. Section III presents the overall architecture and workflow of LAPSE, explaining how fault-free proofs are systematically extended to fault-aware ones. Section IV demonstrates the practicality of the framework through several security-critical case studies considering instruction skip faults, illustrating both manual and automated proof strategies. Section V details the design and implementation of LAPSE, including its fault simulation semantics and proof automation tactics. Finally, Section VI discusses the limitations of the current approach, potential extensions to other fault models, and lessons learned from our case studies.

II. BACKGROUND

A. Rocq

Rocq [15] is an interactive theorem prover based on the Calculus of Inductive Constructions (CiC), a dependently-typed higher-order logic that unifies programs, specifications, and proofs within the Gallina programming language. Dependent types enable mathematical propositions to be parametrized by program values, allowing for precise specifications that relate machine states, traces, and invariants. For our purposes, dependent types also allow proofs to annotate untyped binary values in native programs with dependently-typed propositions to facilitate the proof process.

Bit Widths	$w : \mathbb{N}$
Expressions	$e \coloneqq v \mid n_{\boxed{w}} \mid e_1[e_2]_{\boxed{w}} \mid e_1[e_2 \coloneqq e_3]_{\boxed{w}} \mid e_1 \circledcirc e_2$ $\mid (e : w) \mid \mathsf{let}\ v \coloneqq e_1\ \mathsf{in}\ e_2 \mid *_{\boxed{w}}$
Statements	$s \coloneqq \cdot \mid v \coloneqq e \mid \texttt{jmp}\ e \mid \texttt{exn}\ n \mid s_1; s_2$ $\mid e\ ?\ s_1 : s_2 \mid s^e$
Contexts & Stores	$c, \sigma : \Sigma \coloneqq v \rightharpoonup \mathbb{N}$
Exits	$\chi \coloneqq \downarrow$ (fallthru) $\mid a$ (destination address) $\mid \uparrow i$ (hardware exception)
Programs	$p : \Sigma \to \mathbb{N} \rightharpoonup s$
Traces	$\tau \coloneqq \langle \chi, \sigma \rangle \mid \langle \chi, \sigma \rangle :: \tau$

Fig. 1: Picinæ IL syntax

Users construct proofs in Rocq by incrementally refining *goals*, or *obligations*, using *tactics* that encode steps of logical reasoning. Tactics transform proof states by manipulating assumptions, performing symbolic reasoning, or invoking automated solvers for specific tasks. Completed Rocq proofs are passed to its core proof-checker, which ensures that all goals spawned from manipulating the initial theorem statement have been proved down to the foundations of CiC.

Rocq supplies a rich module type system, allowing users to construct generic units of formalization and proof automation tactics. Specific formalizations and tactics can be generated from these generic descriptions without having to manually re-implementing them. This capability enables us to build generic fault simulations and proof automation tactics that can be instantiated for any Picinæ-supported ISA.

B. Picinæ

Picinæ is a platform in Rocq for instruction-level analysis of executables. It affords machine-verification of any code property expressible in CiC, including properties that describe a program's functional correctness, relating the state of the machine at the program's entry point to the state at its exit points. Machine states are *stores*—partial functions from real and abstract cpu elements to natural numbers. Machine instructions are represented in Picinæ intermediate language (PIL), which is similar to the intermediate languages of other reverse-engineering and binary analysis tools [16], [17]. Programs are functions from a store and virtual address to a PIL *statement* and its size in bytes. This representation supports both Harvard architectures and von Neumann architectures capable of reasoning over self-modifying code.

Figure 1 shows the syntax of PIL. Instructions consist of statements and *expressions*. Statements represent state update and control transfers, while expressions represent the results of instruction-level computations. Expressions read machine state variables v, return binary constants n of a given bit width w, read and write length-w values to/from memory, evaluate standard binary arithmetic and logical operations $\circledcirc$, cast

```
list_tail:
    beqz a0, done   # hd = NULL ? goto done

loop:
    lw   a1, 4(a0) # a1 = cur->next

    beqz a1, done   # nxt = NULL ? found tail

    mv   a0, a1     # cur = nxt

    j loop          # continue loop

done:
    ret             # return cur
```

(a) Instruction skip-vulnerable routine

```
list_tail:
    beqz a0, done   # hd = NULL ? goto done
    beqz a0, done
loop:
    lw   a1, 4(a0) # a1 = cur->next
    lw   a1, 4(a0)
    beqz a1, done   # nxt = NULL ? found tail
    beqz a1, done
    mv   a0, a1     # cur = nxt
    mv   a0, a1
    j loop          # continue loop
    j loop
done:
    ret             # return cur
    ret
```

(b) DMR-augmented, instruction skip-tolerant routine

Fig. 2: RISC-V routines to retrieve the tail of a linked list

subexpressions to new bit widths, make scoped assignments to variables (let-expressions), and return arbitrary width-w values $*_{\boxed{w}}$ when modeling behaviors whose results are unspecified or unpredictable on the target architecture. This final expression form makes the IL's operational semantics non-deterministic, allowing for undefined or unpredictable instruction effects, including faults.

Statement forms include no-op ($\cdot$), variable assignments, control-flow transfers, hardware exceptions, sequences, conditionals, and finite repetitions of statements that model machine instructions that contain internal loops (e.g., the `rep` instruction prefix on Intel ISAs). Program executions are represented as *traces*—sequences of store-*exit* pairs. Exits describe where control flows after each program step, and are either a program counter value or a hardware exception with an integer identifier.

Typical Picinæ proofs prove partial correctness of program fragments w.r.t. trace properties using co-inductive reasoning. Theorems specify an *invariant set* and *exit function*, where the exit function defines the extent of the binary program slice being verified. The invariant set is a partial map from traces to propositions they must satisfy. The exit function is a boolean valued function over stores and addresses defining whether the trace has left the code fragment of interest. Each theorem thereby asserts that all in-scope traces reachable from an initial trace satisfy all invariants they reach. Invariants placed at the exit points are postconditions for the verified code fragment.

Picinæ supports a number of both general-purpose and embedded computing architectures, such as x86-64, RISC-V, and ARMv6-8. Each ISA is defined within an `Architecture` module. These modules instantiate Picinæ's formalizations for the target ISA, and generate tactics for symbolic execution and simplification of program states and binary arithmetic. Furthermore, `Architecture` modules may be used as inputs to additional tooling, as discussed in Section V-B.

C. Fault Injection Vulnerabilities and Defenses

Fault injection attacks deliberately induce transient hardware faults to compromise the integrity of software systems. These attacks exploit the gap between a program's intended semantics and its actual behavior under adverse physical conditions. By inducing faults at precise moments during execution, attackers can bypass security checks, reverse-engineer cryptographic secrets, or manipulate control flow to achieve unauthorized outcomes [18], [19], [20], [21], [22]. The effects of a fault injection attack are described by how precisely the attack can be aimed at a specific region of the processor (*spatial precision*), and how precisely the attack can be launched in a specific window of time (*temporal precision*).

Faults can manifest in various forms depending on the attack vector and target hardware. *Instruction skip* faults cause the processor to skip one or more instructions entirely, effectively replacing them with no-ops. Most often these skips are actually corrupted instructions that have no effect on program state, so treating them as skipped instructions is common. *Bit-flip* faults corrupt individual bits in registers, memory, or instruction encodings, potentially altering operands, opcodes, or addresses. *Control-flow* faults redirect execution to unintended program locations by corrupting branch targets or conditions.

Common physical methods for inducing these faults include *voltage glitching*, where brief fluctuations in the processor's power supply disrupt its normal behavior, and *clock glitching*, which violates the timing assumptions the processor makes [23]. Electromagnetic fault injection (EMFI) uses precise pulses of radiation to induce currents in specific circuit regions. EMFI attacks often utilize lasers for even finer spacial and temporal precision, targeting individual transistors or memory cells [24].

While these attack modes vary in cost, precision, and invasiveness, instruction skip faults are among the most commonly observed and exploited attacks in practice [25], [26], [27]. These faults may be induced by an attacker or by harsh environments, such as in nuclear power plants or in outer space, where high levels of background radiation may cause adverse effects in critical systems.

Fault injection attacks are a realistic threat for any system where an adversary has physical influence over the device, or in the aforementioned harsh environments. Embedded systems,

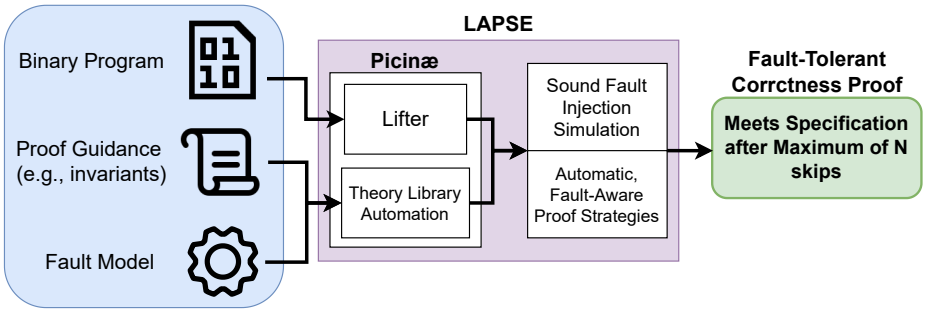


Fig. 3: LAPSE Architecture

IoT devices, smartcards, and secure enclaves are particularly vulnerable due to their deployment in potentially hostile environments [28]. Notable attacks have demonstrated practical breaks of AES [29], RSA [30], and ECC [31].

Due to the prevalence and seriousness of these attacks, protection against fault injection is a well-established area of research with a range of approach angles. Some approaches attempt to add defenses against faults to the program. These defenses can be added during the compilation process [32], or can be applied to the binary after the fact [33]. Alternatives include algorithm-specific defenses [34]. These approaches vary in generalization; some are bound to specific use cases (e.g., specific cryptographic algorithms or fault protections), or are built to work with language-specific compilers. Some methods of applying fault-tolerance defenses to binary programs have been proved to maintain the semantics of the original program under skip fault attacks [35].

Methods for analyzing the fault-tolerance of programs are also common. For example, target codes can be represented as SAT problems [36] or interpreted using symbolic execution [37], [38]. Analysis can also be performed through empirical testing of the target (e.g., [39], [40]), though this has the general disadvantage of incomplete coverage.

D. Formal Binary Analysis.

Recent years have seen a blossoming of binary analysis backed by machine-checked proofs of correctness. Islaris [41] combines SAIL [42] and Isla [43] to create trusted program traces outside of Rocq, then uses Rocq to prove properties about them. IsaBIL [44] embeds BAP's IL [16] in Isabelle/HOL and uses it to reason about program behavior using Hoare logic, effectively extending BAP with Isabelle/HOL's machine-checked front end. A relational logic for unstructured programs in HOL Light reasons about program equivalence and total correctness of assembly codes [45].

E. N-Modular Redundancy

N-Modular Redundancy is a fundamental fault tolerance technique that defends against transient faults by replicating critical computations N times, optionally voting on which output to select, or comparing results to determine whether a fault has occurred. The most common variant is Triple-Modular Redundancy (TMR), which executes a computation three times and uses a majority voting scheme to mask single faults. The computation may or may not be implemented in three distinct ways to limit the ability of persistent faults to disrupt all three results consistently. When one computation produces an incorrect result, the other two out-vote it, allowing the system to continue operating correctly.

Dual-Modular Redundancy (DMR) uses two replicas, optionally comparing the results and jumping to a fault handler if a discrepancy is observed. We consider a specific form of DMR that heavily utilizes duplicated, idempotent instructions. An instruction is idempotent if it has the same result on the program state whether it executes one time or many times. Thus, if all instructions in a program are made to be idempotent and are then duplicated, any number of non-consecutive instruction skip faults can occur without altering the functionality of the program.

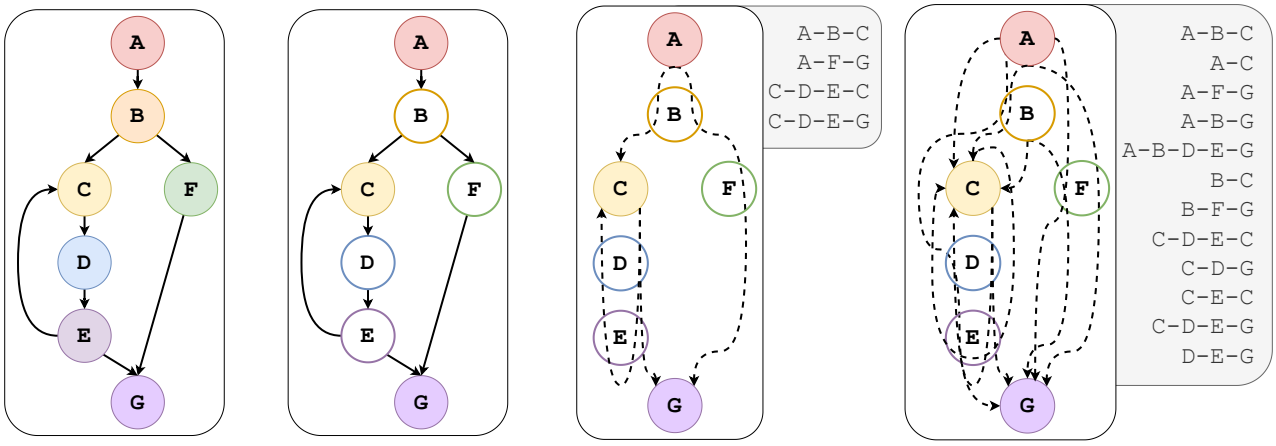
Figure 2 shows an example of this instrumentation on a RISC-V program for finding the tail of a linked list. Each of the original program's instructions are idempotent, so only duplication is required in this case. If they were not, a translation from standard instruction sequences to idempotent sequences could secure the code [33].

III. Overview

A. Workflow

The development of fault-tolerant correctness proofs extends the process of developing standard correctness proofs, requiring semi-automated, formal analysis of program control-flow behavior. The proof development process typically requires two phases: (1) writing a standard correctness proof for the program in a fault-unaware environment, and (2) automating that proof's invariant strategies to handle the many control-flow paths that could arise in a fault-susceptible environment.

Figure 3 shows LAPSE's architecture and workflow. Users provide two inputs: (1) the target binary, and (2) invariant sets. The binary program is automatically lifted into semantically-equivalent PIL. The lifted program is used for symbolic execution and reasoning about the program's behavior in the invariant set and proof. Proofs are facilitated by tactics, theory libraries, and formal definitions of fault mechanics provided by LAPSE and Picinæ.



(a) Program Control Flow (b) Invariant Placement (c) Fault-Unaware Proof Obligations (d) Fault-Aware Proof Obligations

Fig. 4: Relationship between a program's control flow, its invariant placement, and the resulting proof obligations

B. Fault-Free Symbolic Analysis

Correctness proofs in Picinæ perform symbolic interpretation of the target program, exploring all potentially-reachable control-flow paths. Because reachability is undecidable in general, Picinæ conservatively treats all paths as possibly reachable, providing theorems and tactics for eliminating provably unreachable paths from consideration.

Because Picinæ over-approximates control-flow edges, control-flow analysis in LAPSE is a manual task. Users must manually place invariants at appropriate points in the program to describe its CFG.

Symbolic execution occurs within a Rocq proof environment. Picinæ provides lemmas that initiate symbolic interpretation, acting as bridges between Picinæ's over-approximate symbolic interpreter and the correctness specifications of a program. Furthermore, it provides tactics to step the symbolic interpreter forward, spawning goals for new control-flow paths when branches occur.

C. Automatic Fault-Aware Analysis

Extending a Picinæ proof to reason about instruction skips primarily entails automating proof strategies developed in the fault-unaware correctness proof. LAPSE also automatically modifies loop invariants to reason about reachable fault paths.

Figure 4 shows the necessity for automation in proofs of code in fault-susceptible environments. Figures 4a and 4b show a control-flow graph for a program and standard placement of invariant points (program entry and exit, and all join points). Figures 4c and 4d show that the possibility of faults causes a *goal space explosion*, requiring exponentially more proof work. In the worst case this could increase in the the proof space by a factor of n^k, where n is the number of instructions symbolically executed and k is the maximum number of faults to simulate. In a fault-unaware correctness proof, each proof goal verifies a distinct, invariant point-bounded path of execution. In a fault-aware correctness proof, however, each step within such a path has a possibility of faulting a new control-flow path from the current instruction to the next invariant, and so spawns an extra goal.

Our solution to this space explosion problem leverages a key observation: In N-modular redundant programs, the extra proof work turns out to be redundant and memoizable. Based on this realization, we innovate tactics for repeating the original proof strategy at the end of each control-flow branch to solve all extra goals automatically.

Finally, a prerequisite to constructing fault-aware correctness proofs is augmenting Picinæ's ISA semantics with two virtual registers: the Fault Counter (FC) and Fault Timer (FT). These track (1) the number of faults possible throughout the remainder of symbolic execution, and (2) the number of instructions that have successfully executed since the last fault, respectively. These are typically transparent to users of the framework, and used purely internally to model fault behavior; they are never accessed or assigned by target programs. The virtual registers manifest in proofs as proof variables that model fault states.

D. Model Assumptions

LAPSE assumes an execution model in which the processor abides by the ISA subject to the fault model, and assumes an ISA in which multi-threading, preemption, speculative execution, out-of-order execution, and superscalar effects do not affect ISA-intended behavior or contribute to faults. Our example threat model allows single-instruction skips at all instructions. Each skip omits exactly one instruction and advances control flow as if the instruction were absent.

Arbitrary program-counter corruption or other multi-instruction skips are presently out of scope, and are discussed in Section VI. A multi-instruction skip is distinct from multiple single-instruction skips; the former is a single event, for example a program-counter corruption, that skips several

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

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

Fig. 5: `FaultModel` for environments that cause ≤ 1 skip

instructions at once, while the latter consists of several events that each skip one instruction.

IV. Case Studies

We present three examples of skip-fault-aware correctness proofs in Lapse. Each is compiled with `gcc` for 32-bit RISC-V with the I, M, and D extensions enabled. Following compilation, the programs were fully or partially hand-DMR-instrumented as described in Section II-E, and assembled to machine code.

Lapse is not limited to reasoning about DMR-augmented programs. Any fault tolerant program can potentially be verified by our framework. Furthermore, Lapse's approach of IL modification can be generalized to more complex fault-susceptible environments, such as those targeting variable-width instruction ISAs.

A. OpenSSL Cryptographic Memory-compare

We first examine `CRYPTO_memcmp`, a constant-time memory comparison routine from OpenSSL [46]. This routine is extensively utilized in MAC verification and TLS [47], and is therefore a high-priority target for attackers to inflict access control violations. We present two fault-tolerant correctness proofs for this routine, one with a naïve architecture showing that the routine is correct for up to one fault, and another with an automated architecture showing that the routine is correct for up to two non-consecutive faults.

Lapse requires that users define a `FaultModel` module that specifies the maximum number of skips that may occur (`max_faults`), and the minimum number of non-faulted instructions that must execute before another fault is possible (`fault_spacing`). Figure 5 shows that for this proof, we set `max_faults` to 1 and `fault_spacing` to 0. Instantiating this module generates automation for simulating faults and reducing proof effort for fault-aware reasoning.

Equation (1) shows the correctness specification for `CRYPTO_memcmp`:

$$\texttt{return}\ 0 \iff (\forall i, i < \texttt{len} \implies \texttt{in\_a}[i] = \texttt{in\_b}[i]) \quad (1)$$

The target program accepts two pointers `in_a` and `in_b` and a length argument `len`, and returns 0 if and only if the contents of the two pointers are equal for the first `len` bytes.

Figure 6 show the two proofs we present for this program, with some details elided for brevity. Figure 6a shows our proof that the DMR-augmented routine satisfies this specification in a fault-unaware environment. Figure 6b shows the automated fault-aware proof, formatted to show the syntactic similarity to the original.

The key invariant states that after k loop iterations, register `a0` $= 0$ if and only if the input arrays `in_a`, `in_b` are equivalent for their first k bytes. This invariant is proved two times: (1) after starting at the entry point and reaching the loop, and (2) after starting at the loop, not satisfying the break condition, and reaching the loop again. The third main proof case gives the user the postcondition as an obligation after k symbolically interpreted loop iterations. Because the invariant and postcondition are similar, the reasoning strategies for each are nearly identical, requiring a few small lemmas regarding notions of k-equality as used in the invariant, and the behavior of logical OR and XOR as used in the update to register `a0`.

From the fault-unaware correctness proof for DMR-augmented `CRYPTO_memcmp`, we next define a fault model, adapt the lifted IL and invariant set, and automate a proof of correctness for an environment in which up to one instruction skip may occur. The adaptations concern the fault counter (FC) and fault timer (FT) proof variables introduced in Section III-C. The mechanics of the FC and FT are discussed in Section V, but for now it suffices to know that each step of the symbolic interpreter spawns two new control-flow paths, one in which the FC has been decremented if it was nonzero, and one in which it remains unchanged.

To add faults to the computation model, we pass the lifted IL representation of `CRYPTO_memcmp` to the `inject_skips` function, which implements fault simulation in the symbolic interpreter. Next, we augment the precondition invariant with the assumption `fault_assumptions`, which automatically encodes information about the values of FC and FT based on the defined fault model. Each following internal invariant gains the statement `fault_invs`, also encoding information about FC and FT necessary to continue fault simulation.

The proof for the DMR-augmented `CRYPTO_memcmp` shows that it still satisfies the correctness specification in Equation (1) after a maximum of 1 instruction skip at any point in the program. The proof's architecture is intentionally naïve in comparison to the following example in that it always advances the symbolic interpreter until all control paths reach an invariant or postcondition, and then dispatches automatic solvers. The separator lines in Figure 6 show the similarities between the fault-aware and fault-unaware proofs; the automatic solvers directly encode the proof strategies developed in the fault-unaware proof.

Both the fault-unaware and fault-aware proofs consist of 30 lines of definitions and 60 lines of main proof code, and were completed by an expert user in one hour and in 15 minutes, respectively. Both rely on roughly 30 lines of lemmas.

B. BearSSL Conditional Memory-copy

Our second case study is `br_ccopy`, a constant-time conditional memory copy routine from BearSSL [48]. This routine is used in implementations of elliptic curve cryptography in which constant-time table lookups are necessary [49].

```
Proof.
  apply prove_invs. (* begin co-induction *)

  (* Entry -> Loop Invariant *)
  destruct PRE as (A0 & A1 & A2 & Mem). {
  step.
```

```
  - (* len = 0 *)
    repeat step. split; intro. lia.
  - (* len <> 0 *)
    repeat step. lia.
    exists 0. psimpl; repeat split; lia.
  }

  (* Loop Inv -> Loop Inv / Exit *)
  destruct PRE as (k & Mem & A5 & A1 & A2
                   & Bound & Eq). {
  repeat step.
```

```
  - (* solve loop invariant *)
    exists (k + 1). psimpl.
    repeat split; try lia.
    -- unfold k_equal. intros.
       apply or_xor_zero in H.
       destruct H.
       assert (i < k \/ i = k) by lia.
       destruct H2.
         now apply Eq.
         now subst.
    -- intros.
       pose proof (k_equal_inv _ _ _ _ H).
       apply Eq in H0.
       now rewrite H, N.lxor_nilpotent, H0.
```

```
  - (* solve postcondition *)
    replace len with (1 + k) by lia.
    split; intro.
    -- destruct (or_xor_zero H).
       rewrite H in *.
       apply k_equal_step.
        now apply Eq. now rewrite H0.
    -- replace (s R_A0) with 0. psimpl.
       apply N.lxor_eq_0_iff.
       symmetry. apply H. lia.
       apply k_equal_inv in H.
       apply Eq in H. now rewrite H.
  }
Qed.
```

(a) Manual fault-unaware proof

```
Proof.
  apply prove_invs. (* begin co-induction *)

  (* Entry -> Loop Invariant *)
  destruct PRE as (A0 & A1 & A2 & Mem).
```

```
  repeat step;
    (split; intro; lia) ||

    (exists 0; psimpl; repeat split; lia).
```

```
  (* Loop Inv -> Loop Inv / Exit *)
  destruct PRE as (k & Mem & A5 & A1 & A2
                   & Bound & Eq).
```

```
  Local Ltac solve_inv :=
    exists (k + 1); psimpl;
    repeat split; [lia|lia|solve_fault_invs
    | unfold k_equal; intros X i Z;
        apply or_xor_zero in X;
        destruct X;
        assert (i < k \/ i = k) as Y by lia;
        destruct Y;
          [now apply Eq
          |now subst]
    | intros;
       pose proof (k_equal_inv H);
       apply Eq in H0;
       now rewrite H, N.lxor_nilpotent, H0].
```

```
  Local Ltac solve_post :=
    replace len with (1 + k) by lia;
    split; intro;
    [destruct (or_xor_zero H);
     rewrite H in *;
     apply k_equal_step;
      [now apply Eq | now rewrite H0]
    |replace (s R_A0) with 0; psimpl;
     apply N.lxor_eq_0_iff;
     symmetry; apply H; lia;
     apply k_equal_inv in H;
     apply Eq in H; now rewrite H].
  repeat step; (solve_inv || solve_post).
Qed.
```

(b) Automated fault-aware proof

Fig. 6: Correctness proofs for `CRYPTO_memcmp`. Separator lines highlight their nearly-identical structure.

Equation (2) shows the correctness specification for `br_ccopy`:

$$\begin{cases} \texttt{ctl} = 0 & \texttt{mem} = \texttt{mem}' \\ \texttt{ctl} = 1 & \forall i, i < len \implies \texttt{dst}[i] = \texttt{src}[i] \end{cases} \tag{2}$$

The routine accepts two pointers, `dst` and `src`, a length argument `len`, and a control argument `ctl`. If `ctl` is zero, memory is preserved throughout function execution, otherwise the first `len` bytes of `src` are copied into `dst`. Similarly to the previous example, we verify that the DMR-augmented routine satisfies this specification both in a fault-unaware environment and in an environment threatening up to one single-instruction skip.

This fault-aware proof uses LAPSE's automatic step tactic, `eager_step`, to utilize a more careful strategy that limits goal space explosion. Assuming the skip fault has not yet occurred, each step of the symbolic interpreter spawns two control flow paths: one in which the skip occurs, and one in which it does not. After making each step, the already-faulted path is fully traversed until an invariant or postcondition is reached and then solved. Ignoring control flow path splits caused by conditional branches, this limits the number of symbolic execution goals kept in memory at any time to 2. The

invariant and postcondition solvers used in the fault-unaware proof took an expert roughly 15 minutes to construct.

The fault-unaware correctness proof consists of 30 lines of definitions and 80 lines of main proof code, completed by an expert user in roughly one hour. The fault-aware correctness proof consists of 30 lines of definitions and 90 lines of main proof code, completed by an expert user in roughly 30 minutes. Both rely on roughly 30 lines of lemmas.

C. Triple-Modular Redundant Password Checker

Our third case study is `TMR`, a triple-modular redundant password checker utilizing `CRYPTO_memcmp`. This routine runs `CRYPTO_memcmp` three times and utilizes voting to error-correct and produce a result. Its correctness under single-instruction skips depends on its voter: given results A_1, A_2, A_3, it returns[1] $A_1 + A_2 + A_3 \leq 1$. Unlike the previous examples, `TMR` is not completely DMR-augmented, and performs function calls. Return instructions in `CRYPTO_memcmp`, however, have been duplicated to protect against fall-throughs into other code.

`TMR`'s correctness specification is nearly identical to that of `CRYPTO_memcmp` (except returning `true` rather than 0 if the passwords match). We verify that `TMR` satisfies this specification both in a fault-unaware environment and in an environment threatening up to one single-instruction skip. This proof is substantially more difficult than the previous two and offers insight into proofs of fault tolerance at larger scales.

Firstly, the previous two examples study leaf node functions in the call-graph, significantly reducing the complexity of their analysis. The problem of analyzing interprocedural binary code is known to be harder [50], as it involves either trusting or proving that the system ABI is followed by the callee, that control does not leave the boundaries of the callee, etc. In the case of DMR-style hardening, call instructions in a caller cannot merely be duplicated on ISAs with architectural calling conventions that involve register overloading. In RISC-V, the argument register `a0` is also used for storing function return values, so duplicating `call` instructions is unsound.

Secondly, the fault-tolerance of the callee (proved in Section IV-A) does not in this case rely on DMR code idioms; therefore interprocedural flows cannot rely on a DMR-based proof of fault tolerance. For example, even though `CRYPTO_memcmp`'s function body is proven fault-tolerant, a fault at its return instruction could fail to return its (correct) result to the caller. To secure it, we therefore duplicate each of its internal return instructions and ensure the final two instructions are returns.

Because `CRYPTO_memcmp` is not hardened using DMR, the invariant set for `TMR` must track each potential case of whether faults already occurred, and if not, whether a fault occurs within the callee. Table I shows how these cases compound as more calls to `CRYPTO_memcmp` are added. The new cases are outside the scope of some of `eager_step`'s

[1] `CRYPTO_memcmp` accumulates differences in its two input pointers via an 8-bit logical-or, so $A_1 + A_2 + A_3$ can never overflow and satisfy the 32-bit ≤ 1 comparison despite the pointers being non-equal.

Call 1	Call 2	Call 3	Error Corrected
✓	✓	✓	✓
	✓	✓	✓
✓		✓	✓
✓	✓		✓

TABLE I: `CRYPTO_memcmp` fault cases

```
1  Definition inject_skips (p : program)
2                          (s : store)
3                          (a : addr) :=
4    match p s a with
5    | None => None
6    | Some (sz, instr) =>
7        Some (sz,
8          If (fault_spacing < FT &&
9              0 <? FC && Unknown)
10         Then
11           FC := FC - 1;
12           FT := 0
13         Else
14           FT := FT + 1;
15           instr)
16   end.
```

Fig. 7: Simplified definition of `inject_skips`

automation capabilities, and therefore require careful proof engineering to traverse the set of possibilities.

Picinæ's machinery for reasoning about function calls significantly reduces the size of the correctness proof, saving approximately 700 lines of proof code. The resulting `CRYPTO_memcmp` proof consists of roughly 60 lines of proof code in a fault-unaware context and 200 lines in a fault-aware context, both completed by an expert user in roughly one hour. The proof for `TMR` consists of roughly 260 lines of proof code in a fault-unaware context and 400 lines in a fault-aware context, both completed by an intermediate user in 12 hours.

V. System Design and Implementation

Lapse is implemented as a set of architecture-agnostic definitions, theorems, and tactics in Rocq 8.19.2. The system can be easily instantiated for new architectures with similar semantic domains (registers, memory, etc.). Its faculties for fault simulation consist of 60 lines of definitions and 50 lines of theorems, and 10 lines of definitions and theorems and 75 lines of tactics for fault aware proof automation.

A. Simulation

Figure 7 shows the definition of `inject_skips` (with smaller details elided for clarity). This function simulates single-instruction skips by wrapping each instruction's IL sequence in a non-deterministic branch. On line 4, the IL of the instruction at address `a` in program `p` is retrieved. Lines 5 and 6 determine whether an instruction exists at this address, and if so, wraps it in the condition on line 8.

This condition checks whether enough steps have passed for a fault to happen (if the FC is non-zero), and further branches on an `Unknown` value (denoted $*_{\boxed{1}}$ in Figure 1). The check against `Unknown` introduces non-determinism in the symbolic interpreter. The interpreter expects its value to be a conditional

```
1  Ltac es step solver :=
2    (time step; revgoals);
3    [> match goal with
4      | [|- nextinv _ _ _ _ _] =>
5             es step solver
6      | _ => clean_fault_goals;
7             time (try solve [solver])
8      end ..].
9
10 Tactic Notation
11     "eager_step" tactic(arch_step)
12     "by" tactic(Solver) :=
13   (* look for fault assumptions / invs *)
14   (try process_faults);
15   (* recursive careful step *)
16   (es arch_step Solver).
```

Fig. 8: `eager_step` tactic definition

(either 0 or 1), and because it cannot be determined, proof obligations for both are generated. This results in one goal in which the instruction is replaced by decrementing FC and clearing FT (lines 11-12) and one in which FC does not change, FT is incremented, and the instruction operates as normal (lines 14-15).

B. Automation

LAPSE provides several automation tactics. Figure 8 shows the most prominent, `eager_step`, the memory-conservative goal space exploration tactic used in automatic fault-tolerance proofs. This tactic scans for assumptions in the proof context (line 13) that reference FC and FT, and uses that information to prepare the goal for symbolic execution. It then launches symbolic execution and sorts the goals in order of increasing FC (line 2). Line 3 selects each of these goals individually and either continues stepping (line 5) or attempts to solve an invariant or postcondition with a given solver tactic (line 7).

By selecting and solving goals individually, `eager_step` often substantially reduces the system's maximum memory usage throughout the proof. A cursory test shows that just by replacing `repeat step` with `eager_step` in the fault-aware proof in Figure 6b, 3.6GB of memory are consumed as opposed to the prior 4.4GB. This 20% decrease in memory consumption becomes substantial as the number of faults permitted by the user's fault model increases due to the polynomial-factor increase in goals generated by each step.

Furthermore, `eager_step` tolerates any modification to the proof's accompanying `FaultModel`, such as changing the maximum number of faults or the spacing between them, as long as the modified fault model does not break the correctness of the function. To demonstrate, we provide an additional proof in which `CRYPTO_memcmp` is shown to be correct for up to 2 non-consecutive instruction skips. This proof is identical to the one shown in Figure 6b, only using `eager_step` instead of `repeat step`. Thus, the proof generation logic of one fault-aware proof can be reused to generate proofs for multiple non-breaking faults.

This automation is enclosed in a `FaultTolerance` functor that takes as input a Picinæ `Architecture` module describing ISA semantics, allowing it to be applied to programs of any architecture. When extending a Picinæ ISA definition for use in LAPSE, virtual registers that store FC and FT must be added. The user binds these virtual registers to the values of FC and FT by instantiating a `FaultToleranceConfig` module, an input to `FaultTolerance`. For our RISC-V implementation of LAPSE, the register modification consists of a single line change, and the instantiation of the `FaultToleranceConfig` and `FaultTolerance` modules consists of 20 lines of code.

VI. DISCUSSION

A. Memory Corruption

Future work can extend LAPSE to reason about memory corruption faults through non-deterministic corruption simulation, similarly to how it reasons about instruction skip events. At each step, the symbolic executor branches on whether a corruption occurs, conditionally assigning abstract, unconstrained values to memory locations before executing an instruction. This approach naturally fits within LAPSE's existing fault simulation framework.

Realistically modeling the semantics of memory corruption events is an important challenge for such future work. A model that corrupts all memory at every execution step is overly pessimistic, since it excludes nearly all programs that use memory. Effective formalization therefore requires careful choice of constraints that reflect realistic fault models while maintaining proof tractability.

One such choice is to provide a spatial bound to memory corruption events in a similar manner to how LAPSE already defines bounds on instruction skip occurrences and spacing. If the fault model admits memory corruption of at most n consecutive bytes per fault, programs that maintain redundant copies of data structures larger than n bytes, or separated by at least n bytes, can be verified. Because no single corruption event can affect both copies, dynamic consistency checks become feasible and potentially verifiable. This approach is particularly suitable for $n \leq 8192$, as modern DRAM rows are usually 8K in length [51], so corruption events larger than 8K require additional reasoning and formalization of the physical memory layout.

Another LAPSE-compatible approach for simulating memory corruption explicitly assumes that consecutive steps of symbolic execution do not exhibit differences in memory that are known to be very low-probability events in real adversarial environments. For example, assuming that a specific memory region will not remain corrupted throughout consecutive program steps enables verification of programs that read from memory multiple times to ensure that the values read into registers match the values that exist in memory.

B. Multiple-Instruction Skip Faults

Our current prototype focuses on single-instruction skip faults, which are the most typical skip-faults in practice. Future work can support multiple-instruction skip faults by augmenting the fault-aware ISA semantics with a Multi-Instruction

Skip Counter (MISC). This counter is zeroed until a fault occurs, after which it takes any value up to `mskip_max_length`, which is configured in the user's `FaultModel`. As long as MISC is non-zero, the current instruction does not execute, and MISC is decremented. This generalizes single-instruction skip models by setting `multi_skip_max_length` $= 1$.

The main challenge is a corresponding increase in proof space size, since each step generates `mskip_max_length` extra goals, which grows much faster than the single-instruction skip methodology if `max_faults` is larger than 1. Future work should address this by considering code hardening strategies similar to DMR that yield redundancies in these extra proof goals, allowing proof automation to solve the goals by coalescing the work into common subgoals or lemmas.

C. Alternative Forms of Faults

Specific alternative fault categories are not suitable for formal reasoning due to goal space explosions that dwarf those seen in skip fault proofs. For example, naïvely simulating register corruption by assigning random values to any register could produce up to several hundreds of new goals at each symbolic step. Many of these new goals might have had return addresses and stack pointers corrupted, a corruption many threat models do not include [52]. Not included in this work for similar reasons are condition flag corruption and instruction skips induced by modification of the program counter.

However, load and store faults could be simulated and handled in a very similar manner to LAPSE's treatment of skips. Programs that read out of duplicate memory ranges and compare results to detect faults could also be verified using this technique.

D. Skip Fault Side-Effects

Instruction skips are a special case of the more general fault model in which an attacker can corrupt instructions before or during decoding [53]. These corrupted instructions often effectively skip the original instruction because the corrupted instruction is semantically equivalent to a no-op—it does not affect program values that are relevant to the program's correctness. Our demonstrations of LAPSE therefore do not include all potential side-effects in its simulation of skip-like faults. Future work can extend LAPSE to reason about instruction corruption faults by corrupting the instructions of the binary program and decoding them with Picinæ's decoders.

E. Automation

Althgouh LAPSE and Picinæ provide automation facilities for symbolic execution, type-safe lifting of binaries to IL, fault simulation and reasoning, and fixed-width binary arithmetic solving, constructing fault-aware correctness proofs still requires non-trivial manual effort. The examples in Section IV show that the manual effort required to complete fault-aware correctness proofs is on the order of hours per routine.

This manual effort arises in different tasks depending on details of the program being analyzed. For the DMR-augmented programs, the majority of manual effort was devoted to constructing the fault-unaware correctness proof. Formal proofs of binary functional correctness are known to be difficult to construct, requiring expert knowledge of each component of the program and corresponding mathematical skill for translating these to the automated theorem prover. The process of converting these proofs to fault-aware proofs was significantly easier in comparison.

For `TMR`, however, the majority of the manual effort was in constructing the fault-aware correctness proof. This is both due to overall target code complexity and its non-uniform DMR augmentation, which makes it substantially more difficult to reason about the behavior of its calls to `CRYPTO_memcmp` than in the fault-unaware proof. Developing logical predicates that generalize the descriptions of `CRYPTO_memcmp`'s correctness under all placements of faults within the function would solve these issues and give way to more general fault-aware automation, and is envisioned for future work.

VII. CONCLUSION

We presented LAPSE, a framework for constructing machine-checked correctness proofs for native code in the presence of arbitrarily many single-instruction skip faults. LAPSE performs fault-aware symbolic execution by wrapping a program's intermediate representation in a non-deterministic construct that may skip an instruction. This non-determinisim forces the user to reason about instruction skips at each potential skip site, ensuring that any invariant or postcondition respects all fault possibilities. Users may configure the parameters of the fault simulation by adjusting values in a `FaultModel` module.

We evaluated LAPSE on security-critical routines from OpenSSL and BearSSL, as well as a triple-modular-redundant password checker. The proofs for these routines showed that the development of fault-aware correctness proofs often requires only a trivial amount of modification of a standard correctness proof. LAPSE automated much of this process via the `eager_step` tactic, which comprehensively traverses all potential control-flow paths and limits memory consumption.

LAPSE provides a foundation for richer fault simulation techniques via the patterns set in `inject_skips` and `FaultModel`. We discuss methods for extending the framework to reason about bounded memory corruption and multi-skip fault events. Finally, scaling these techniques to larger systems and more complex interprocedural code remains an important future research direction, particularly for mission-critical systems where physical fault threats are of significant concern. Overall, LAPSE demonstrates that formal verification need not assume idealized hardware to remain tractable. By bringing fault injection directly into the scope of machine-checked proofs, this work moves formal methods closer to the realities faced by secure and dependable systems deployed in adversarial or harsh environments.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation CISE Graduate Fellowships under Grant No. 231998.

REFERENCES

[1] K. Fisher, "HACMS: High assurance cyber military systems," *ACM SIGAda Ada Letters*, vol. 32, no. 3, pp. 51–52, 2012.

[2] M. Bozzano, H. Bruintjes, A. Cimatti, J.-P. Katoen, T. Noll, and S. Tonetta, *Formal Methods for Aerospace Systems*. Springer Singapore, 2017, pp. 133–159.

[3] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM (CACM)*, vol. 52, no. 7, p. 107â€"115, 2009.

[4] J. Guan, H. Li, X. Li, X. Wang, B. Wang, Q. Wang, S. Qin, M. He, M. Armanuzzaman, and Z. Zhao, "Formally verifying the state machine of TLS 1.3 handshake in OpenSSL," in *Proceedings of the 44th IEEE Conference on Computer Communications (INFOCOM)*, 2025.

[5] S. Berezin, "Model checking and theorem proving: A unified framework," Ph.D. dissertation, Carnegie Mellon University, 2002.

[6] D. Wheeler, "How to prevent the next Heartbleed," https://dwheeler.com/essays/heartbleed.html, April 2024.

[7] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 207–220.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *Proceedings of the 42nd ACM Symposium on Principles Of Programming Languages (POPL)*, 2014, pp. 179–191.

[9] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[10] E. P. Kim and N. R. Shanbhag, "Soft n-modular redundancy," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 323–336, 2010.

[11] D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet, "Fault injection for formal testing of fault tolerance," *IEEE Transactions on Reliability*, vol. 45, no. 3, pp. 443–455, 1996.

[12] A. Serrano-Cases, J. Isaza-González, S. Cuenca-Asensi, and A. Martínez-Álvarez, "On the influence of compiler optimizations in the fault tolerance of embedded systems," in *IEEE 22nd International Symposium on On-line Testing and Robust System Design (IOLTS)*, 2016, pp. 207–208.

[13] K. W. Hamlen, D. Fisher, and G. R. Lundquist, "Source-free machine-checked validation of native code in Coq," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2019, pp. 25–30.

[14] F. Verbeek, J. A. Bockenek, Z. Fu, and B. Ravindran, "Formally verified lifting of C-compiled x86-64 binaries," in *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2022, pp. 934–949.

[15] Rocq, "Rocq: A trustworthy, industrial-strength interactive theorem prover and dependently-typed programming language for mechanised reasoning in mathematics, computer science and more," https://rocq-prover.org, 2025.

[16] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011, pp. 463–469.

[17] C. D. B. Knighton, "Ghidra – journey from classified NSA tool to open source," Black Hat, 2019.

[18] T. Chiu and W. Xiong, "SoK: Fault injection attacks on cryptosystems," in *Proceedings of the 44th IEEE Symposium on Security & Privacy (S&P)*, 2023, pp. 64–72.

[19] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert, "One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization," in *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 2875–2889.

[20] Z. Kazemi, A. Papadimitriou, I. Souvatzoglou, E. Aerabi, M. M. Ahmed, D. Hely, and V. Beroulle, "On a low cost fault injection framework for security assessment of cyber-physical systems: Clock glitch attacks," in *Proceedings of the IEEE 4th International Verification and Security Workshop (IVSW)*, 2019, pp. 7–12.

[21] X. M. Saß, R. Mitev, and A.-R. Sadeghi, "Oops..! I glitched it again! How to multi-glitch the glitching-protections on ARM TrustZone-M," in *Proceedings of the 32nd USENIX Security Symposium*, 2023, pp. 6239–6256.

[22] C. Bozzato, R. Focardi, and F. Palmarini, "Shaping the glitch: Optimizing voltage fault injection attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pp. 199–224, 2019.

[23] J. Breier and X. Hou, "How practical are fault injection attacks, really?" *IEEE Access*, vol. 10, pp. 113 122–113 130, 2022.

[24] M. A. Elmohr, H. Liao, and C. H. Gebotys, "EM fault injection on ARM and RISC-V," in *Proceedings of the 21st International Symposium on Quality Electronic Design (ISQED)*, 2020, pp. 206–212.

[25] V. S. Nguyen, V. Grosso, and P.-L. Cayrel, "Practical persistent fault attacks on AES with instruction skip," *IACR Communications in Cryptology*, vol. 2, no. 1, 2025.

[26] A. Adiletta, M. C. Tol, K. Derya, B. Sunar, and S. Islam, "LeapFrog: The rowhammer instruction skip attack," in *Proceedings of the IEEE 10th European Symposium on Security and Privacy (EuroS&P)*, 2025.

[27] Espressif, "Espressif security advisory concerning fault injection and secure boot (CVE-2019-15894)," https://www.espressif.com/en/news/Espressif_Security_Advisory_Concerning_Fault_Injection_and_Secure_Boot, September 2019.

[28] A. Gangolli, Q. H. Mahmoud, and A. Azim, "A systematic review of fault injection attacks on IoT systems," *Electronics*, vol. 11, no. 13, p. 2023, 2022.

[29] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of AES," in *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2012, pp. 7–15.

[30] A. Sidorenko, J. van den Berg, R. Foekema, M. Grashuis, and J. de Vos, "Bellcore attack in practice," *IACR Cryptology ePrint Archive*, p. 553, 2012.

[31] K. Schneider, L. Auer, and A. Wagner, "Fault attacks on ECC signature verification," *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2025, no. 4, pp. 1010–1052, 2025.

[32] B. Pesin, S. Boulmé, D. Monniaux, and M.-L. Potet, "Formally verified hardening of C programs against hardware fault injection," in *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2025, pp. 140–155.

[33] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[34] M. Christofi, B. Chetali, L. Goubin, and D. Vigilant, "Formal verification of a CRT-RSA implementation against fault attacks," *Journal of Cryptographic Engineering (JCEN)*, vol. 3, no. 3, pp. 157–167, 2013.

[35] J. Richter-Brockmann, A. R. Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, "FIVER – robust verification of countermeasures against fault injections," *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, pp. 447–473, 2021.

[36] H. Tan, P. Gao, F. Song, T. Chen, and Z. Wu, "SAT-based formal verification of fault injection countermeasures for cryptographic circuits," *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)*, vol. 2024, no. 4, pp. 1–39, 2024.

[37] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation (ICST)*, 2014, pp. 213–222.

[38] S. Ducousso, S. Bardin, and M.-L. Potet, "Adversarial reachability for program-level security analysis." in *Proceedings of the 32nd European Symposium on Programming (ESOP)*, 2023, pp. 59–89.

[39] V. Khuat, J.-M. Dutertre, and J.-L. Danger, "Software countermeasures against the multiple instructions skip fault model," *Microelectronics Reliability*, vol. 155, p. 115370, 2024.

[40] S. Patranabis, A. Chakraborty, and D. Mukhopadhyay, "Fault tolerant infective countermeasure for AES," in *Proceedings of the International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, 2015, pp. 190–209.

[41] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell, "Islaris: Verification of machine code against authoritative ISA semantics," in *Proceedings of the 43rd ACM Conference on Programming Language Design and Implementation (PLDI)*, 2022, pp. 825–840.

[42] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proceedings of the 46th ACM-SIGACT Symposium*

on Principles of Programming Languages (POPL), no. POPL, pp. 71:1–71:31, 2019.

[43] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell, "Isla: Integrating full-scale ISA semantics and axiomatic concurrency models (extended version)," *Formal Methods in System Design*, vol. 63, no. 1, pp. 110–133, 2024.

[44] M. Griffin, B. Dongol, and A. Raad, "IsaBIL: A framework for verifying (in)correctness of binaries in Isabelle/HOL," in *Proceedings of the 39th European Conference on Object-oriented Programming (ECOOP)*, vol. 333, 2025, pp. 14:1–14:30.

[45] D. Mazzucato, A. Mohamed, J. Lee, C. Barrett, J. Grundy, J. Harrison, and C. S. Păsăreanu, "Relational Hoare logic for realistically modelled machine code," in *Proceedings of the 37th International Conference on Computer Aided Verification (CAV)*, 2025, pp. 389–413.

[46] OpenSSL, "CRYPTO_memcmp," https://docs.openssl.org/3.3/man3/CRYPTO_memcmp, 2026.

[47] G. Moshkin and D. Ivanov, "OpenSSL: tls_common.c," GitHub, 2025.

[48] T. Pornin, "BearSSL," https://bearssl.org, 2018.

[49] ——, "BearSSL: Why constant-time crypto?" https://bearssl.org/constanttime.html, 2018.

[50] J. Bockenek, F. Verbeek, and B. Ravindran, "Exceptional interprocedural control flow graphs for x86-64 binaries," in *Proceedings of the 21st International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2024, pp. 3–22.

[51] T. M. O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proceedings of the 16th USENIX Security Symposium*, 2007.

[52] F. Q. Yuan, "Formal framework and tools to derive efficient application-level detectors against memory corruption attacks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.

[53] H. Jiang, X. Zhu, and J. Han, "Instruction-fetching attack and practice in collision fault attack on AES," *Symmetry*, vol. 14, no. 10, p. 2201, 2022.

```
CRYPTO_memcmp:
    beq     a2,zero,.L4
    mv      a5,a0
    add     a2,a0,a2
    li      a0,0
.L3:
    lbu     a3,0(a5)
    lbu     a4,0(a1)
    xor     a4,a4,a3
    or      a0,a0,a4
    addi    a5,a5,1
    addi    a1,a1,1
    bne     a5,a2,.L3
    ret
.L4:
    li      a0,0
    ret
```

Fig. 9: CRYPTO_memcmp

APPENDIX

Figures 9 and 10 show the RISC-V assembly code for the CRYPTO_memcmp routine from OpenSSL. The former is generated by GCC, and the latter has been manually DMR-instrumented to resist non-consecutive faults.

Figures 11 and 12 show the RISC-V assembly code for the br_ccopy routine from BearSSL. These programs are again generated by GCC and manually DMR-instrumented, respectively.

Figure 13 shows the RISC-V assembly code for the TMR routine. The DMR-augmented form of this routine replaces non-idempotent instruction sequences with idempotent ones, and duplicates each instruction in TMR except for call instructions. The CRYPTO_memcmp subroutine is unchanged in the DMR-augmented form of the program except for duplicated ret instructions.

```
CRYPTO_memcmp:
        beq     a2,zero,.L4
        beq     a2,zero,.L4
        mv      a5,a0
        mv      a5,a0
        add     t0,a0,a2
        add     t0,a0,a2
        mv      a2,t0
        mv      a2,t0
        li      a0,0
        li      a0,0
.L3:
        lbu     a3,0(a5)
        lbu     a3,0(a5)
        lbu     a4,0(a1)
        lbu     a4,0(a1)
        xor     t0,a4,a3
        xor     t0,a4,a3
        mv      a4,t0
        mv      a4,t0
        or      t0,a0,a4
        or      t0,a0,a4
        mv      a0,t0
        mv      a0,t0
        addi    t0,a5,1
        addi    t0,a5,1
        mv      a5,t0
        mv      a5,t0
        addi    t0,a1,1
        addi    t0,a1,1
        mv      a1,t0
        mv      a1,t0
        bne     a5,a2,.L3
        bne     a5,a2,.L3
        ret
        ret
.L4:
        li      a0,0
        li      a0,0
        ret
        ret
```

Fig. 10: Fault-Tolerant CRYPTO_memcmp

```
br_ccopy:
    neg     a0,a0
    add     a6,a1,a3
    beq     a3,zero,.L1
.L3:
    lbu     a4,0(a1)
    lbu     a5,0(a2)
    addi    a1,a1,1
    addi    a2,a2,1
    xor     a5,a4,a5
    and     a5,a5,a0
    xor     a5,a5,a4
    sb      a5,-1(a1)
    bne     a1,a6,.L3
.L1:
    ret
```

Fig. 11: br_ccopy

```
br_ccopy:
    neg     t0,a0
    neg     t0,a0
    mv      a0,t0
    mv      a0,t0
    add     a6,a1,a3
    add     a6,a1,a3
    beq     a3,zero,.L1
    beq     a3,zero,.L1
.L3:
    lbu     a4,0(a1)
    lbu     a5,0(a2)
    lbu     a4,0(a1)
    lbu     a5,0(a2)
    addi    t0,a1,1
    addi    t0,a1,1
    mv      a1, t0
    mv      a1, t0
    addi    t0,a2,1
    addi    t0,a2,1
    mv      a2, t0
    mv      a2, t0
    xor     t0,a4,a5
    xor     t0,a4,a5
    mv      a5, t0
    mv      a5, t0
    and     t0,a5,a0
    and     t0,a5,a0
    mv      a5, t0
    mv      a5, t0
    xor     t0,a5,a4
    xor     t0,a5,a4
    mv      a5, t0
    mv      a5, t0
    sb      a5,-1(a1)
    sb      a5,-1(a1)
    bne     a1,a6,.L3
    bne     a1,a6,.L3
.L1:
    ret
    ret
```

Fig. 12: Fault-Tolerant `br_ccopy`

```
CRYPTO_memcmp:
    beq     a2,zero,.L4
    mv     a5,a0
    add     a2,a0,a2
    li     a0,0
.L3:
    lbu     a3,0(a5)
    lbu     a4,0(a1)
    xor     a4,a4,a3
    or     a0,a0,a4
    addi     a5,a5,1
    addi     a1,a1,1
    bne     a5,a2,.L3
    ret
.L4:
    li     a0,0
    ret


TMR:
    addi     sp,sp,-32
    sw     ra,28(sp)
    sw     s0,24(sp)
    sw     s1,20(sp)
    sw     s2,16(sp)
    sw     s3,12(sp)
    sw     s4,8(sp)
    mv     s0,a0
    mv     s1,a1
    mv     s2,a2
    call     CRYPTO_memcmp
    mv     s3,a0
    mv     a2,s2
    mv     a1,s1
    mv     a0,s0
    call     CRYPTO_memcmp
    mv     s4,a0
    mv     a2,s2
    mv     a1,s1
    mv     a0,s0
    call     CRYPTO_memcmp
    snez     a5,s3
    snez     a4,s4
    add     a5,a5,a4
    snez     a0,a0
    add     a0,a5,a0
    sltiu     a0,a0,2
    lw     ra,28(sp)
    lw     s0,24(sp)
    lw     s1,20(sp)
    lw     s2,16(sp)
    lw     s3,12(sp)
    lw     s4,8(sp)
    addi     sp,sp,32
    jr     ra
```

Fig. 13: `TMR`