ProofFrog → EasyCrypt

Douglas Stebila, University of Waterloo

Formosa Project retreat • May 27, 2026

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

https://prooffrog.github.io • https://www.douglas.stebila.ca/research/presentations/

What is ProofFrog?

A tool for checking transitions in introductory-level game-hopping proofs.

Domain-specific language for primitives / schemes / games / proofs, C/Java-like syntax.

Focus on ease of use at the expense of expressivity.

Engine: Automated canonicalization

Proof author specifies reductions inducing sequence of games.

Two adjacent games are **interchangeable** iff they **canonicalize** to the same AST under a fixed pipeline of ~82 automated **transforms.**

Examples of transforms:

- Inline Single-Use Variable
- Topological Sort
- Merge/Split Uniform Samples
- XOR Cancellation
- RF → uniform

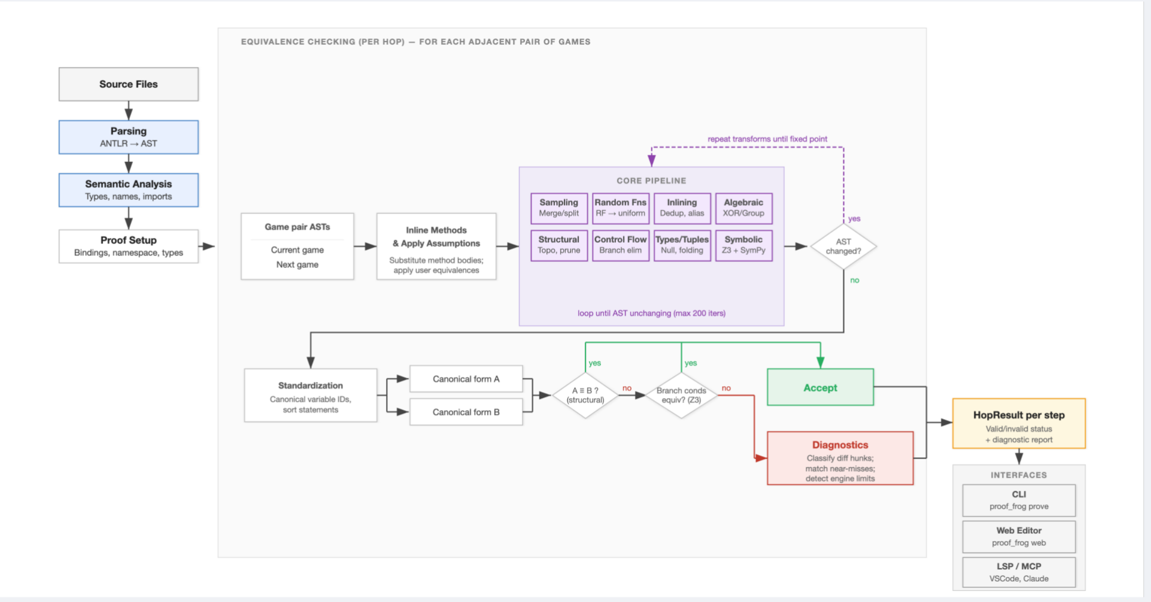
Use of **Z3** / **SymPy** to evaluate some conditions

If canonicalization equivalence fails, no manual override.

Trust Assumptions

Entire code base is the trust assumption.

1. Assume the 82 transforms preserve semantics.
2. Assume the Z3 calls are sound.
3. Assume the implicit FrogLang semantics match the implementation.



Source Files
Parsing
ANTLR → AST
Semantic Analysis
Types, names, imports
Proof Setup
Bindings, namespace, types
EQUIVALENCE CHECKING (PER HOP) — FOR EACH ADJACENT PAIR OF GAMES
Game pair ASTs
Current game
Next game
Inline Methods
& Apply Assumptions
Substitute method bodies;
apply user equivalences
repeat transforms until fixed point
CORE PIPELINE
Sampling
Merge/split
Random Fns
RF → uniform
Inlining
Dedup, alias
Algebraic
XOR/Group
Structural
Topo, prune
Control Flow
Branch elim
Types/Tuples
Null, folding
Symbolic
Z3 + SymPy
loop until AST unchanging (max 200 iters)
yes
AST
changed?
no
Standardization
Canonical variable IDs,
sort statements
Canonical form A
Canonical form B
yes
A ≡ B ?
(structural)
no
yes
Branch conds
equiv? (Z3)
no
Accept
Diagnostics
Classify diff hunks;
match near-misses;
detect engine limits
HopResult per step
Valid/invalid status
+ diagnostic report
INTERFACES
CLI
proof_frog prove
Web Editor
proof_frog web
LSP / MCP
VSCode, Claude

Highlight: draft-irtf-cfrg-hybrid-kems-10

4 KEM combiners:

	hash everything	hash like X-Wing
group + KEM	**UG**	**CG**
KEM + KEM	**UK**	**CK**

x 2 for seeded versus expanded secret keys

Target properties:

- correctness
- IND-CCA assuming alg 1 secure
- IND-CCA assuming alg 2 secure
- LEAK-BIND-K-{CT,PK}
- HON-BIND-K-{CT,PK}

Formalization in ProofFrog

- 44 mechanized proofs:
  - correctness · IND-CCA-PQ · IND-CCA-T · LEAK-BIND-K-{CT,PK} · HON-BIND-K-{CT,PK}
- most proofs in standard model, 2 in ROM (CCA under DH for CG)
- random oracle proofs rely on statistical helper game hops
- **novel observation:**
  - CG & CK seeded couldn't be proven LEAK-BIND in standard model
  - showed them LEAK-BIND in ROM and HON-BIND in standard model
- proofs developed mostly by Claude, scaffolded from some similar examples
  - about 20 hours of time (including necessary engine extensions)

EC exporter design: per–transform, not per–hop

Per–hop

- Export one lemma with reductions per game → game step
- Try to construct EC tactics reflecting the entire canonicalization chain

Per–transform

- Export one lemma with reductions **for every transform applied within each ProofFrog game canonicalization**
- Many more lemmas exported to EC, but each covering a much smaller diff

Supplement each ProofFrog transform with EC export strategy:

1. **static pre–programmed tactic script**
2. **parametric** pre–programmed tactic script
3. **interactive** per–proof discovery:
   - Cache an EC proof for this particular step of this particular proof
   - Could be human written or LLM–generated via EasyCrypt MCP server
4. admit.

Progress so far

7 proofs!

Successfully exporting 7 basic ProofFrog proofs (including schemes and games) to EasyCrypt, fully verified in EasyCrypt, including:

- one-time-pad is one-time secure (real/random)
- one-time-pad is one-time secure (left/right)
- chained encryption is one-time secure
- **length-doubling PRG => length-tripling PRG**

Transforms

- **Canned (static):** 15 transforms
  - Inline Single-Use Variable → proc; sp; auto.
  - Symbolic Computation / Normalize Commutative Chains → proc; sim.
- **Canned parametric:** 4 transforms
  - Python synthesizes a per-instance tactic.
  - Uniform XOR Simplification, Merge/Split Uniform Samples, swap synthesizers for Topological Sort / Bubble Sort.
- **Interactive:** 12 transforms
  - No automated tactic script
  - Closed by hand/LLM then cached. e.g. Sink Uniform Sample, Merge Product Samples, XOR Cancellation.
- **Open** / admit.**:** 51 transforms
  - No recipe yet. Many are RF / RO transforms, tuple expansion, group transforms.

Worked example: TriplingPRG_PRGSecurity

Scheme: T(s) = G(s)[0..λ] ‖ G(G(s)[λ..2λ])

ProofFrog proof

https://github.com/ProofFrog/examples/blob/easycrypt/Proofs/PRG/TriplingPRG_PRGSecurity.proof

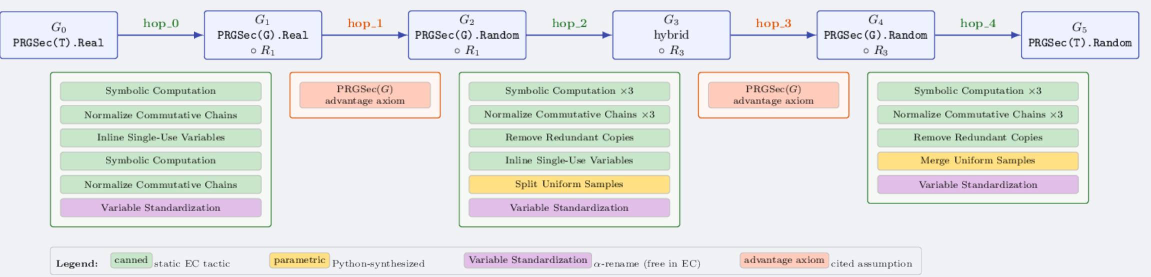
- 5 hops with 2 user-specified reductions
- 54-line file

Exported EasyCrypt proof

https://github.com/ProofFrog/examples/blob/easycrypt/Proofs/PRG/TriplingPRG_PRGSecurity.ec

- 5 hops leading to 48 intermediate game states
- 7 unique transforms applied ~25 times in total
- 74 lemmas
- 1662-line file

Worked example: TriplingPRG_PRGSecurity



Worked example: TriplingPRG_PRGSecurity

Theorem. For T(s) = G(s)[0..λ] ‖ G(G(s)[λ..2λ]) stretching G : λ → 2λ to T : λ → 3λ:

$$| \Pr[\text{Game_step_0}] - \Pr[\text{Game_step_5}] | \le 2 \cdot \text{eps_PRGSecurity}$$

via 5 hops with explicit reductions R1, R3.

BitString types bs_{λ,2λ,3λ}

- is_lossless / is_funiform / is_full axioms
- xor, slice, concat, the slice ∘ concat = id axioms
- decomposition of uniform sampling dmap (d × d)

abstract theory PRG_Theory:

- Scheme / Oracle / Adv module types
- Game_PRGSecurity_{Real,Random}
- bound eps_PRGSecurity
- assumption axiom PRGSecurity_advantage.

Two clones:

- G_c (λ → 2λ)
- T_c (λ → 3λ)

Scheme

TriplingPRG (G : G_c.Scheme) : T_c.Scheme.

section Main:

- main reductions R1, R3
- six Game_step_i wrappers

Worked example: TriplingPRG_PRGSecurity

■ For each interchangeability hop i ∈ {0,2,4}, **four layers:**

1. **Two chains** Step_iL_state_0..N, Step_iR_state_0..M for each side of hop; one module per canonicalization step
2. **One** micro_* **lemma per transform:**
   - static canned / parametric canned / interactively developed & cached
3. **One** canon_bridge_i **lemma** connecting terminal LHS and terminal RHS (proc; sim.)
4. **One** hop_i **lemma** gluing the chains for the entire hop together via transitivity

▒ **Assembling the bound:**

- **Interchangeability hops:** Apply the lemmas from above: byequiv … proc; call (_: ={glob G}); first by conseq hop_i; auto.
- **Assumption hops** (hop_{1,3}_pr): rewrite each Pr[Game_step_k] into Pr[G_c.Game_PRGSecurity_*(G, R{k}_Adv(G,A))], then apply PRGSecurity_advantage.
- **Main theorem:** have the five hops, smt(G_c.eps_PRGSecurity_pos). Adversary carries {-G}.

▒ **Status.** Zero admit.! 🎉

Curiosities mapping ProofFrog → EasyCrypt

▓▓▓ Bitstrings of generic length.

- FrogLang has BitString<n> parametric in arbitrary integer expressions (λ, 2λ, 3λ, Nseed, …).
- EC has no generic bitstring type — we generate one abstract bs_<n> per distinct length expression with axiomatised slice_* / concat_* / xor_* and a lossless funiform dbs_<n>.

▓▓▓ Length-expression canonicalization.

- λ+λ vs 2*λ: must render as the same EC string or sim can't match. We pipe every integer argument through SymPy. Fragile.

▓▓▓ Adversary footprint.

- ProofFrog assumes primitives are stateless functions.
- Translate that into a {-G} adversary restriction into EasyCrypt.

Role of LLMs

I've used **Claude Code** aggressively across the stack:

ProofFrog engine development

- Most engine developments over the past 3 months are LLM-implemented with human guidance.

Authoring ProofFrog proofs via a `prooffrog` MCP server.

- MCP commands let LLM see intermediate game state after any transform to diagnose failures
- LLM was able to extract proof for StarFortress (Deidre Connolly & Paul Grubbs) direct from PDF
- Most of the CFRG-hybrid-KEMs proofs were written by the LLM, based on existing examples catalog but no further guidance on proof strategy

EasyCrypt exporter development.

- All of the EC exporter (translators, tactic synthesizers, cache) are LLM-implemented
- Significant benefit from access to `easycrypt-mcp` server for generic transformer tactic script and per-proof tactics
  - `cli_open` / `cli_step` / `cli_print` over a persistent EC process

What's next?

ProofFrog proofs

DDH/CDH/DLOG relations

IND definitions with bitstring length

Attack counterexamples

ProofFrog functionality

LaTeX export

Bound calculation

Clean-up hybrid/induction arguments

ProofFrog in classes

Current trial at TUE (Andreas & Kathrin) and FAU (Paul Rösler)

Incorporate feedback & revise

Prepare package for instructors

What's next for ProofFrog → EasyCrypt?

Validate current approach

Check that the schemes, games, and lemmas being exported are meaningful.

Will there be scalability problems with the per-transform export strategy?

Mapping ProofFrog data types on to EC data types

- BitString<lambda>

Tricky transforms

Canonicalization steps involving random sampling

Lazy/eager random function re-sampling

What could help

LLM-oriented documentation:

- purpose & use of each tactic
- catalog of re-usable lemmas
- catalog of proof examples with common strategies

Links

ProofFrog info

ProofFrog website (*https://prooffrog.github.io*)

Eprint 2025/418 (*https://eprint.iacr.org/2025/418*)

GitHub repositories

Engine (*https://github.com/ProofFrog/ProofFrog*)

Examples (*https://github.com/ProofFrog/examples*)

Case study: CFRG hybrid KEMs

https://github.com/ProofFrog/examples/tree/easycrypt/applications/cfrg-hybrid-kems

EasyCrypt export

https://github.com/ProofFrog/examples/blob/easycrypt/Proofs/PRG/TriplingPRG_PRGSecurity.ec

Appendix: BitString<λ> export

```
(* Abstract type + uniform distribution *)
type bs_λ.
op  dbs_λ : bs_λ distr.
axiom dbs_λ_ll   : is_lossless dbs_λ.
axiom dbs_λ_fu   : is_funiform dbs_λ.
axiom dbs_λ_full : is_full     dbs_λ.

(* XOR as an abstract group operation *)
op xor_λ : bs_λ -> bs_λ -> bs_λ.
axiom xor_λ_invol  : forall a b, xor_λ (xor_λ a b) b = a.
axiom xor_λ_commut : forall a b, xor_λ a b = xor_λ b a.

(* Concat / slice, mangled by source+target widths *)
op concat_bs_λ_bs_λ_to_bs_2λ : bs_λ -> bs_λ -> bs_2_λ.
op slice_bs_2λ_to_bs_λ       : bs_2_λ -> int -> int -> bs_λ.

axiom concat_slices_id : forall (s : bs_2_λ),
  concat_bs_λ_bs_λ_to_bs_2λ
    (slice_bs_2λ_to_bs_λ s 0      λ)
    (slice_bs_2λ_to_bs_λ s λ (λ + λ)) = s.

(* The payoff: split a wide uniform sample into two independent ones *)
axiom dbs_2_λ_split : dbs_2_λ =
  dmap (dbs_λ `*` dbs_λ)
       (fun p => concat_bs_λ_bs_λ_to_bs_2λ p.`1 p.`2).
```

Appendix: PRG theory

```
abstract theory PRG_Theory.
  type bs_λ_t.
  type bs_λ_stretch_t.
  op   dbs_λ_t : bs_λ_t distr.
  op   dbs_λ_stretch_t : bs_λ_stretch_t distr.
  ...
  module PRGSecurity_Real (G : Scheme) :
PRGSecurity_Oracle = {
    proc query() : bs_λ_stretch_t = {
      var s : bs_λ_t;  var _r0 :
bs_λ_stretch_t;
      s   <$ dbs_λ_t;
      _r0 <@ G.evaluate(s);
      return _r0;
    }
  }.
  ...
  op eps_PRGSecurity : real.
  axiom PRGSecurity_advantage (Em <: Scheme)
  (A <: PRGSecurity_Adv) &m :
    `| Pr[Game_PRGSecurity_Real(Em, A).main()

          @ &m : res]
       - Pr[Game_PRGSecurity_Rand(Em, A).main()

          @ &m : res]
     | <= eps_PRGSecurity.
```

```
clone PRG_Theory as G_c
with
  type bs_λ_t
         <- bs_λ,
  type bs_λ_stretch_t
        <- bs_2_λ,
  op   dbs_λ_t
         <- dbs_λ,
  op   dbs_λ_stretch_t
         <- dbs_2_λ.

clone PRG_Theory as T_c
with
  type bs_λ_t
         <- bs_λ,
  type bs_λ_stretch_t
         <- bs_3_λ,
  op   dbs_λ_t
         <- dbs_λ,
  op   dbs_λ_stretch_t
         <- dbs_3_λ.
```

Appendix: Tripling PRG scheme

```
module TriplingPRG (G : G_c.Scheme) : T_c.Scheme = {
  proc evaluate(s : bs_λ) : bs_3_λ = {
    var result1, result2 : bs_2_λ;
    var x, y : bs_λ;
    result1 <@ G.evaluate(s);
    x       <- slice_bs_2_λ_to_bs_λ result1 0      λ;
    y       <- slice_bs_2_λ_to_bs_λ result1 λ (2*λ);
    result2 <@ G.evaluate(y);
    return concat_bs_λ_bs_2_λ_to_bs_3_λ x result2;
  }
}.
```

Appendix: Per-transform micro lemmas

```
(* transform: Normalize Commutative Chains (bucket=canned) *)
lemma micro_0_left_2 :
  equiv [ Step_0L_state_2(G).query ~ Step_0L_state_3(G).query :
          ={glob G} ==> ={res, glob G} ].
proof. proc; sp; wp; sim. qed.
```

```
(* transform: Symbolic Computation (bucket=canned) *)
lemma micro_0_left_3 :
  equiv [ Step_0L_state_3(G).query ~ Step_0L_state_4(G).query :
          ={glob G} ==> ={res, glob G} ].
proof. proc; sp; wp; sim. qed.
```

```
(* transform: Variable Standardization (bucket=canned) *)
lemma micro_0_left_5 :
  equiv [ Step_0L_state_5(G).query ~ Step_0L_state_6(G).query :
          ={glob G} ==> ={res, glob G} ].
proof. proc; sp; wp; sim. qed.
```

Appendix: Chaining micro lemmas via transitivity

```
lemma hop_0_chain :
  equiv [ Step_0L_state_0(G).query ~ Step_0R_state_0(G).query :
          ={glob G} ==> ={res, glob G} ].
proof.
  transitivity Step_0L_state_1(G).query
    (={glob G} ==> ={res, glob G}) (={glob G} ==> ={res, glob G});
    [ smt() | smt() | apply micro_0_left_0 | ].
  transitivity Step_0L_state_2(G).query (...) (...);
    [ smt()|smt()|apply micro_0_left_1 |].
  ...
  transitivity Step_0R_state_6(G).query (...) (...);
    [ smt()|smt()|apply canon_bridge_0 |].
  transitivity Step_0R_state_5(G).query (...) (...);
    [ smt()|smt()|apply micro_0_right_5_rev |].
  ...
  apply micro_0_right_0_rev.
qed.
```

Appendix: Hop probability lemma

```
lemma hop_1_pr (A <: T_c.PRGSecurity_Adv {-G}) &m :
  `| Pr[Game_step_1(G, A).main() @ &m : res]
   - Pr[Game_step_2(G, A).main() @ &m : res] | <= G_c.eps_PRGSecurity.
proof.
  have hL : Pr[Game_step_1(G, A).main() @ &m : res]
          = Pr[G_c.Game_PRGSecurity_Real(G, R1_Adv(G, A)).main() @ &m : res]
    by byequiv => //; proc; inline *; sim.
  have hR : Pr[Game_step_2(G, A).main() @ &m : res]
          = Pr[G_c.Game_PRGSecurity_Random(G, R1_Adv(G, A)).main() @ &m : res]
    by byequiv => //; proc; inline *; sim.
  rewrite hL hR.
  apply (G_c.PRGSecurity_advantage G (R1_Adv(G, A)) &m).
qed.
```

Appendix: Final theorem

```
lemma main_theorem (A <: T_c.PRGSecurity_Adv {-G}) &m :
  `| Pr[Game_step_0(G, A).main() @ &m : res]
   - Pr[Game_step_5(G, A).main() @ &m : res] |
   <= G_c.eps_PRGSecurity + G_c.eps_PRGSecurity.
proof.
  have h0 := hop_0_pr A &m.
  have h1 := hop_1_pr A &m.
  have h2 := hop_2_pr A &m.
  have h3 := hop_3_pr A &m.
  have h4 := hop_4_pr A &m.
  smt(G_c.eps_PRGSecurity_pos).
qed.
```