# Formal Verification of a Typical Smart Contract for the InnoChain Blockchain

A. V. Krasnenkova

***RTU MIREA***, ***RusBITech-Astra***, *Moscow, Russian Federation*

**SpacSec 2026,**
**ITMO University**

[26 — 28 October 2026]



# The Problem: Smart Contract Security

Blockchain smart contracts manage billions of dollars in digital assets.

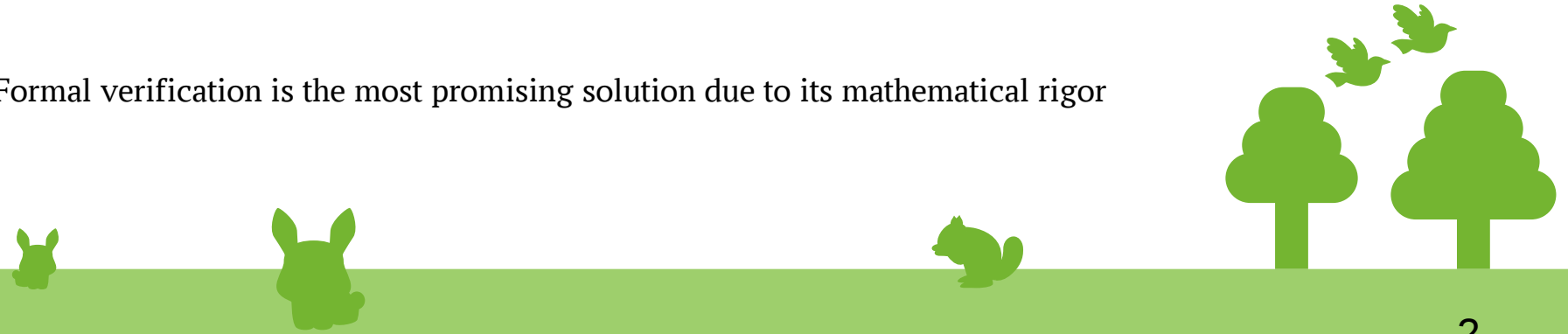
Their immutable nature means bugs cannot be patched after deployment.

Traditional testing and code audits cannot completely guarantee the absence of vulnerabilities.

Real-world incidents:

- The DAO hack (2016): $60 million lost;
- Parity Wallet (2017): $280 million frozen;
- Reentrancy attacks remain a critical threat.

→ Formal verification is the most promising solution due to its mathematical rigor



# InnoChain: The First Formally Verified Russian Blockchain

**Key principles:**

**1. CakeML Integration:**

- Contracts written in CakeML — a formally verified subset of Standard ML;
- Fully formal semantics within HOL4 theorem prover.

**2. HOL4 Theorem Prover:**

- Interactive proof environment;
- Rich library of mathematical theories.

**3. Monadic Execution Model:**

- `monadChainScript.sml` framework;
- Trace-based reasoning with **LTL**.

**→ End-to-end verification: from source code to machine code execution**



# Why Formal Verification?

| Approach | Guarantee |
| --- | --- |
| Unit testing | Covers specific cases only |
| Code audit | Hihly depends on human expertise |
| **Formal verification** | Mathematical proof for almost all scenarios |

**Goal:** Mathematically prove that a smart contract behaves exactly as intended — under any execution scenario.



# Case Study: "Tour Operator Trip Order"

**Contract States:**

**OrderCreated → OrderApproved → FlightApproved**

**→ FlightConfirmed → OrderCompleted**

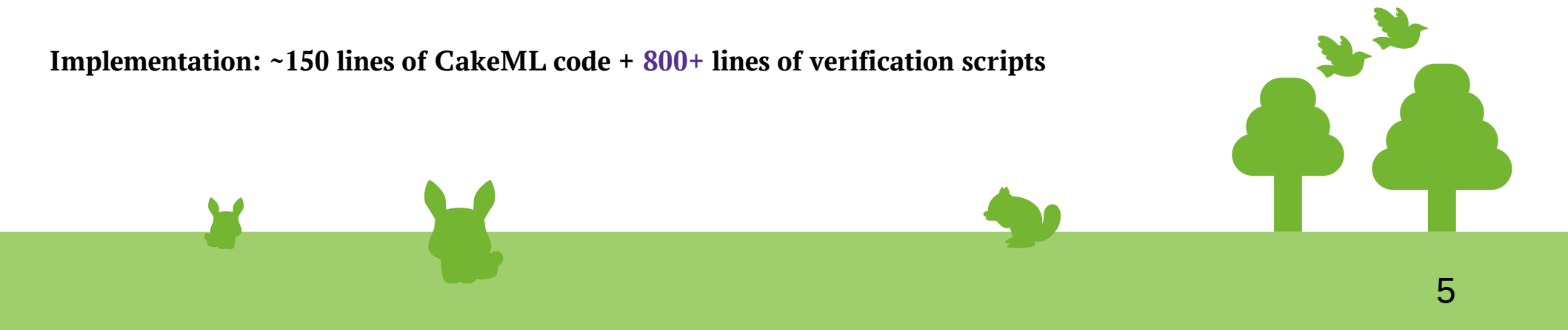
↘️ **OrderRejected (can be reached from OrderApproved, FlightApproved, FlightConfirmed)**

**Business Functions (8 functions, which govern the overall ordering process):**

**approveOrder, setFlight, approveFlight, bookTickets,**

**createPayment, confirmPaidFlight, setRoute, confirmRouteAndFlight**

**Implementation: ~150 lines of CakeML code + 800+ lines of verification scripts**



# Three-Level Verification Methodology

| Level | Approach | Properties |
|---|---|---|
| 1. | **Temporal (LTL-based)** | Terminal state invariance, reachability |
| 2. | **Structural-Logical** | State immutability, monotonicity, reachability |
| 3. | **Deductive Verification** | Necessary/sufficient conditions |

**Key idea:** Different techniques complement each other and provide independent confirmation of critical properties



# Level 1: Temporal Property Verification (LTL-based)

**Framework**: `monadChainScript.sml` (developed by K. V. Ziborov, **Positive Technologies**)

**Key components**:

- `Environment` — global state including contract code and states;
- `ChainTrace` — reflexive-transitive relation over transaction sequences;
- **LTL** operators — for temporal reasoning.

**Monadic composition:**

```
do
    r <- f params;
    s <- get_state
od
```

→ Proving properties that hold across **all** execution traces



# Property 1: Invariance of Rejection

*Once the status becomes **OrderRejected**, it remains unchanged forever*

```
Theorem request_canceled_holds:

∀(e1 e2 : Environment) (s1 s2 : State) caddr.

  get_envContracts e1 caddr = SOME sc4 ∧

  get_envContractStates e1 caddr = SOME s1 ∧

  ...

  ChainTrace e1 e2 ∧

  (get_Request_status s1.request = OrderRejected) ⇒

  (get_Request_status s2.request = OrderRejected)
```

**Proof strategy:**

- **Induction** on ChainTrace;
- **Case analysis** on all possible actions;
- **Function-by-function** verification.



# Property 2: Invariance of Completion

*Once the status becomes **OrderCompleted**, it remains unchanged forever*

```
Theorem request_closed_holds:

∀(e1 e2 : Environment) (s1 s2 : State) caddr.

  get_envContracts e1 caddr = SOME sc4 ∧

  get_envContractStates e1 caddr = SOME s1 ∧

  ...

  ChainTrace e1 e2 ∧

  (get_Request_status s1.request = OrderCompleted) ⇒

  (get_Request_status s2.request = OrderCompleted)
```

**Security guarantees:**

✔️ Irreversibility of final decisions;

✔️ Protection against reuse (no re-calling after completion);

✔️ Behavioral transparency

→ Prevents attacks like "cancellation of confirmed order"



# Level 2: Structural-Logical Approach

***Developed by the author***

**Key idea**: Analyze contract behavior at the function level itself without relying on the monadic trace framework.

**Effective for**:

✔️ Correctness of state updates;

✔️ Conditions for phase transitions;

✔️ Error handling.

**Verified properties**:

- State immutability on error;
- Status monotonicity on success;
- Reachability of rejecting state;
- Status decreasing on rejection



# Property 4: State Immutability on Error

*If an arbitrary function fails, the contract state remains unchanged*

```
Theorem diff_states_must_coincide_for_ALL_ERRORS_fail:

∀ f params prevState nextState error.

  (do r <- execute f params; s <- get_state od

  prevState = return nextState prevState) ∧

  (chooseFunction f params prevState = fail error nextState) ⇒

  (prevState = nextState)
```

**Significance:**

✔️ **Atomicity guarantee** — each function either succeeds or leaves state unchanged;

✔️ **Audit simplification** — no partial changes from exploit attempts;

✔️ **Universal property** — applies to **all** smart contracts

# Property 5: Status Monotonicity on Success

*Under correct sequential execution, the status strictly increases:*

**OrderCreated (1) → OrderApproved (2) → FlightApproved (3)**

**→ FlightConfirmed (4) → OrderCompleted (5)**

```
Theorem status_increasing_for_approveOrder:

∀ state params state_new.

  (HD params = SCBool T) ∧

  (approveOrder params state = (Success SCUnit, state_new)) ⇒

  status_numeration (get_Order_status state_new.order) >

  status_numeration (get_Order_status state.order)
```

**Ensures:**

✔️ Predictable behavior;

✔️ Protection against regression and race attacks;

✔️ Business logic alignment.

⚠️ **Valid for simple linear graphs without cycles**



# Level 3: Deductive Verification

***Developed by the author***

**Key idea**: Prove necessary and sufficient conditions for each function's success or failure.

**Provides**:

✔️ Complete characterization of all error triggers;

✔️ Necessary and sufficient conditions for success;

✔️ Post-conditions for successful execution.

**Example** — `approveOrder` errors:

| Error | Condition |
|---|---|
| `WRN_PARAMS` | Parameter is not Boolean |
| `ORDER_CREATED_ONLY` | `Current status` ≠ **OrderCreated** |
| `TOUR_OPERATOR_ONLY` | Caller is not the **Tour operator** |



# Deductive Verification: Necessary and Sufficient Conditions for Success

**Theorem approveOrder_success:**

```
∀ state params state_new. ∃ decision.
  let status = get_Order_status state.order;
      sender = id (get_context_msgSender state.context);
      tourOperator = (get_Order_tourOperator state.order);
      roleID = (get_Role_id tourOperator);
      ...
  in
  (approveOrder params state = SUCCESS) ⇔
  (checkTypes params [TypeBool]) ∧
  (status = OrderCreated) ∧
  (sender = roleID) ∧
  (∃ v. HD params = SCBool v)
```

**Interpretation:**

***The function succeeds iff:***

- Parameter is a Boolean value;
- Status is **OrderCreated**;
- Caller is the **Tour operator**.







# Verified Properties — Summary

| # | Property | Level |
|---|---|---|
| 1. | Invariance of **OrderRejected** | **1** |
| 2. | Invariance of **OrderCompleted** | **1** |
| 3. | Reachability of all states | **1, 2** |
| 4. | State immutability on error | **2** |
| 5. | Status monotonicity on success | **2** |
| 6. | Reachability of rejecting state | **2** |
| 7. | Error conditions (all types) | **3** |
| 8. | Success conditions (all functions) | **3** |






**Result:** Complete coverage of contract behavior, mathematically proven in HOL4.

# Comparison of Verification Approaches

| Aspect | Monadic (LTL) | Structural-Logical |
|---|---|---|
| **Developed by** | K. Ziborov | The author |
| **Object** | Execution traces | Function logic |
| **Properties** | Trace invariants | Reachability, atomicity |
| **Requires** | Trace model | Per-function analysis |
| **Used for** | Temporal properties | Function behavior |

→ **Integration provides:**

- Complementarity of methods;
- Independent confirmation;
- Deeper property verification.



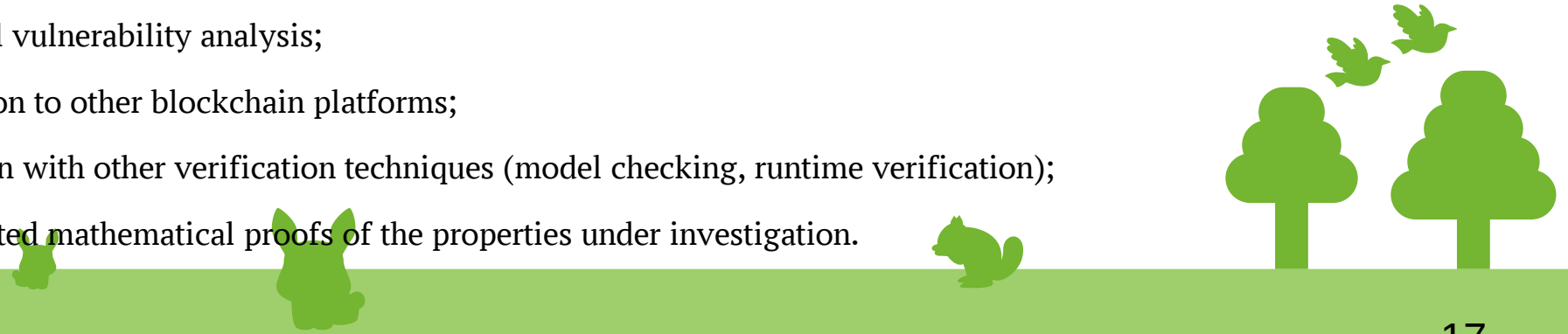
# Conclusion & Future Work

**Contributions:**

- Comprehensive three-level methodology for smart contract verification;
- Two original approaches (structural-logical + deductive verification);
- End-to-end verification of a production-grade contract on InnoChain.

**Future Work:**

- Extending to more complex contract patterns;
- Automation tools for structural-logical approach;
- Independent verification of high-level specifications;
- Low-level vulnerability analysis;
- Application to other blockchain platforms;
- Integration with other verification techniques (model checking, runtime verification);
- Complicated mathematical proofs of the properties under investigation.



# Thank you!

# Questions?

- **Contacts: ankrasnenkova@gmail.com, akrasnenkova@astralinux.ru, @nastya_krass,**

**Github: nastyakrass**

🔗 **Code repository:**

**https://drive.google.com/drive/folders/1DRYLhOuqATKw1wER9b9biJWFsTpqLQ1S**



# Formal verification is funny!

