Knowledge Graph-Based Gas Optimisation for Smart Contracts: Exploratory Vision and Challenges

8th Distributed Ledger Technology Workshop
June 5 2026, Pula, Italy

Giacomo Ibba
Department of Business School, University of Cagliari
giacomof.ibba@unica.it

Gavina Baralla
Department of Mathematics and Computer Science, University of Cagliari
gavina.baralla@unica.it

Introduction

Decentralised applications (DApps) represent one of the most prominent applications of the Ethereum Blockchain.

These systems manage a considerable amount of currency, requiring stringent security requirements and leading to complex implementations.

Network congestion may cost users hundreds of dollars to execute complex operations, limiting accessibility and hindering widespread adoption.

Motivation

Layer 2 solutions and alternative chains offer reduced costs, even though gas efficiency remains a crucial challenge.

Layer 2 networks don’t solve the problem of inefficient contracts accumulating costs at scale.

High gas costs prevent user from interacting with DApps.

Developers may face pressure to sacrifice safety for efficiency.

Considerations on optimisation strategies

Profiling and testing

Application of established patterns

Verification of preserved semantics and readability

Employment of specialised tools

Established patterns

Storage Layout

– Pack vars into 32 bytes slots
– uint128 + uint128 = 1 slot (declared consecutively)
– Mark constants as constant/immutable
– Prefer memory over storage in loops

Loop optimisation

– Cache array.length in a local var
– Use ++i not i++ (in older compiler versions)
– Unchecked{} for safe counters
– Avoid dynamic array growth

Function visibility

– external cheaper than public
– Use calldata for read-only arrays
– Inline short private functions
– Avoid redundant view calls

Short circuits & checks

– require() before state changes
– Cheapest checks first (&&)
– custom errors < string revert
– Reorder conditions by cost

Data types

– Use uint256 not uint8 in isolation
– bytes32 cheaper than string
– Avoid dynamic types in storage
– bool costs full 32-byte slot when declared in isolation

Event & logs

– Logs ~8× cheaper than storage
– Use indexed sparingly (375 gas each)
– Emit instead of store for history
– Batch events where possible

Knowledge Graph

Knowledge graphs (KG) are structured semantic networks that represent entities as nodes and their relationships as edges, enabling rich querying and reasoning capabilities.In the context of smart contract analysis, knowledge graphs offer several advantages over traditional program analysis approaches

The semantic richness of knowledge graphs allows them to capture type information, data flow relationships , and execution semantics that are essential for understanding gas costs. Optimisations can be represented as graph rewriting operations, providing a formal foundation for automated code transformation.

The semantic layer in knowledge graphs enables domain-specific reasoning about Ethereum-specific concepts such as storage locations, state mutability and external call patterns.

This additional semantic information is essential for identifying optimisation opportunities that would be invisible to traditional program analysis tools designed for general-purpose languages.

Triple model and ontology

Triple anatomy

Subject	`st:Function_transfer`
Predicate	`sc:visibility`
Object	`"public"`

As Turtle (.ttl):

```
st:Function_transfer sc:visibility "public" .
```

Ontology layers

OWL Axioms

Disjointness, transitivity, domain/range — logical constraints the reasoner enforces

RDFS Schema

Class hierarchy (subClassOf), property definitions — what types and relations exist

Instances

Named individuals — the actual contract entities extracted by the AST visitor

KG vs relational DB — key differences:

✓ Schema-on-read	✗ Schema-on-write (rigid)
✓ Inference built-in	✗ Manual JOIN logic
✓ Graph traversal (paths)	✗ Recursive CTEs only

Why employ KGs?

What CFG & DFG do not capture

✗ No cross-contract semantics

CFG and DFG operate within a single function or at best a single contract. They cannot express that ContractA inherits a vulnerability from ContractB, or that a library call propagates an access control flaw across a protocol.

✗ No type ontology

Variables in a DFG are storage locations, not semantic entities. There is no concept of "this is an address acting as an owner" — the distinction between uint256 used as a token amount vs a timestamp is invisible.

✗ No queryable schema

CFG/DFG are algorithmic structures: you traverse them in code. There is no standard language to ask "find all public functions that write state without a modifier" across 10,000 contracts simultaneously.

✗ No reasoning or inference

Neither graph type supports logical inference. If sc:inheritsFrom is declared transitive in OWL, a reasoner propagates vulnerabilities up inheritance chains automatically. CFG/DFG require you to implement this traversal manually.

What the Knowledge Graph adds

✓ Protocol-wide semantic scope

Every contract, function, event, and variable across an entire protocol lives in one graph. A single SPARQL query traverses inheritance hierarchies, library dependencies, and cross-contract calls without any bespoke traversal code.

✓ Named, typed entities with ontology alignment

Every node is a named IRI with a declared class. sc:Function_transfer is not just a node — it is an instance of sc:Function, which is a subclass of swo:SoftwareMethod. This alignment links findings to external taxonomies like CWE automatically.

✓ Declarative, reusable audit queries

SPARQL is a standard, composable query language. A pattern written once — "public state-writing function with no modifier" — runs unchanged over any contract in the graph. There is no need to re-implement the detector per contract or per codebase.

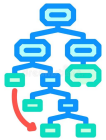
✓ OWL inference propagates facts for free

Declaring sc:inheritsFrom as owl:TransitiveProperty means the reasoner infers deep inheritance chains without storing redundant triples. Vulnerability annotations on a base contract propagate to all children automatically.

Proposed research methodology

1 **Ontology design**

2 **AST extraction**



3 **Triple store ingestion**

4 **Pattern detection**

5 **Gas / security hits**

6 **Audit report**

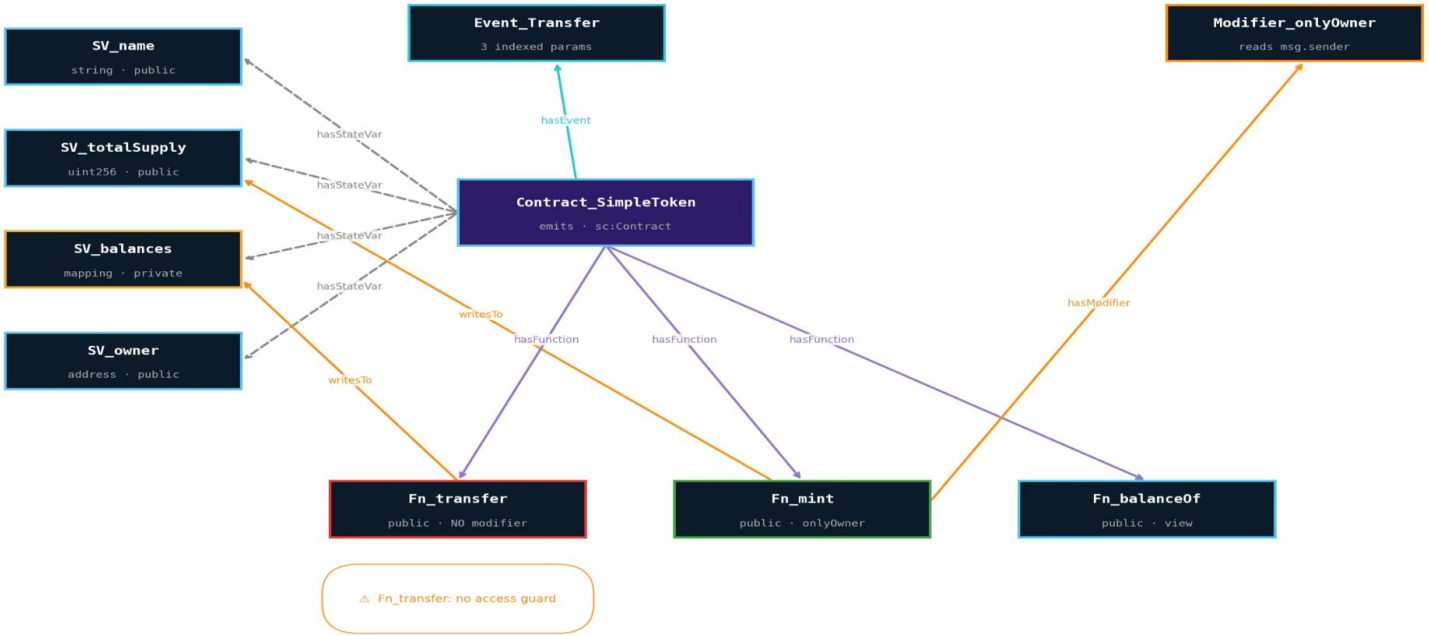
SimpleToken example

```
contract SimpleToken {
  string  public name = "SimpleToken";
  uint256 public totalSupply;
  mapping(addr=>uint256) private bal;
  address public owner;
  event Transfer(address,address,uint);
  modifier onlyOwner() { … }
  function transfer(…) public {
    require(bal[msg.sender]>=amount);
    bal[msg.sender] -= amount;
    bal[to] += amount;
    emit Transfer(…);
  }
  function mint(…) public onlyOwner {
    totalSupply += amount;
  }
  function balanceOf(…) public view {
}
```

← sc:visibility=public
← sc:dataType=uint256
← WARN: pack slots?
← sc:dataType=address
← sc:Event
← sc:Modifier
← sc:visibility=public
← WARN: no modifier guard
← sc:writesTo=balances
← sc:writesTo=balances
← sc:emits=Transfer
← sc:hasModifier
← sc:writesTo=totalSupply
← sc:mutability=view

■ gas risk ■ warning ■ KG entity ■ state mutation

SimpleToken KG



SPARQL Query

```
PREFIX sc: <https://example.org/ontology/sc#>
PREFIX st: <https://example.org/contracts/…>

SELECT ?fn ?contract
WHERE {
  ?contract sc:hasFunction ?fn .
  ?fn  sc:visibility  "public" ;
       sc:mutability  ?mut ;
       sc:writesTo    ?sv .
  FILTER(?mut != "view" && ?mut != "pure")
  FILTER NOT EXISTS {
    ?fn sc:hasModifier ?m .
  }
}
ORDER BY ?contract
```

Namespace prefixes — define sc: and st: shorthands

SELECT — choose variables to return

WHERE block — triple pattern matching

Graph pattern: contract → fn traversal

Literal filter: only public functions

FILTER: exclude view/pure (read-only)

FILTER NOT EXISTS: negative check — no modifier

ORDER BY — sort output deterministically

Result on SimpleToken: → `st:Fn_transfer`

Gas and security queries

Unprotected state-writing public functions

```
SELECT ?fn WHERE {
  ?fn sc:visibility "public" ;
      sc:writesTo ?sv .
  FILTER NOT EXISTS { ?fn sc:hasModifier ?m } }
```

→ Fn_transfer (writes balances, no guard)

Candidate constant state variables (never written)

```
SELECT ?sv ?name WHERE {
  ?sv a sc:StateVariable ; sc:name ?name .
  FILTER NOT EXISTS { ?fn sc:writesTo ?sv } }
```

→ SV_name (never mutated — use constant)

All functions that emit the Transfer event

```
SELECT ?fn WHERE {
  ?fn sc:emits st:Event_Transfer . }
```

→ Fn_transfer, Fn_mint (both emit)

Functions mutating state with payable mutability

```
SELECT ?fn WHERE {
  ?fn sc:mutability "payable" ;
      sc:writesTo ?sv . }
```

→ (none) — SimpleToken has no payable fns

Expected caught patterns

Storage operations on the Ethereum Virtual Machine incur high costs: **SLOAD** (storage read) costs 2,100 gas for cold access and 100 gas for warm access, whilst **SSTORE** (storage write) can cost over 20,000 gas for initial writes.

Contracts that **repeatedly access** the same state variables within a function or loop **without intermediate modifications** present clear optimisation opportunities.

External contract calls require a minimum of 2,600 gas and may be repeated with **identical parameters** within the same **transaction context.**

View and pure functions that return deterministic results based solely on their inputs are amenable to memoisation.

Loop inefficiencies encompass several related patterns. Array length calculations performed on every loop iteration can be **cached** in a local variable before the loop.

Loop-invariant storage reads, where a state variable is accessed inside a loop but never modified within the loop body, can be hoisted **outside the loop.**

Thanks for your attention!