

SolBehave: Evaluating the Behavioural Correctness of LLM-Generated Smart Contracts

Giovanni Pio Delvecchio*, Stefano Fantazzini*, Giacomo Frisoni*, Gianluca Moro*, Lorenzo Bacchiani, Mario Bravetti, and Thomas Troels Hildebrandt

Abstract—Large language models increasingly generate Solidity smart contracts that, once deployed, can never be patched, making a single latent fault permanent and exploitable. Yet current evaluations mostly ask only whether generated code compiles. We introduce SolBehave, a benchmark for evaluating the correctness of model-generated contracts at scale, moving beyond ad hoc unit tests. Across 250 real, reproducible Ethereum contracts, we apply three complementary tests of behavioural fidelity: differential fuzzing, historical-transaction replay, and, as our core contribution, synthetic-transaction replay guided by a Dynamic Condition Response (DCR) graph that enumerates the space of legitimate function-call sequences. Differential fuzzing compares each generated function with the original on automatically generated inputs; historical replay compares the whole contract on the transactions it actually received on-chain; DCR-guided replay compares it on sequences of calls that follow its intended usage, including orderings that never occurred on-chain. We evaluate seven state-of-the-art open-weight models after one round of automated repair. Compilation success ranges from 26.4% to 92.7%, but under differential fuzzing only 31.4% of all generated function bodies that compile, pooled over models and samples, are equivalent to the original. Divergence at the contract level is the norm rather than the exception: for six of the seven models, between 13% and 31% of the generated contracts that compile diverge from the original under historical replay, and between 13% and 44% under DCR-guided replay; only Gemma-4-31B stays below 3% on both. Compilation is therefore a necessary condition for correctness and far from a sufficient one.

Index Terms—smart contracts, large language models, code generation, differential testing, dynamic condition response graphs, Solidity, Ethereum, behavioural fidelity, program synthesis evaluation.

I. INTRODUCTION

Smart contracts, programs deployed on a blockchain, increasingly govern real economic activity, and the stakes of getting them wrong have grown with it. The clearest example is tokenisation, the representation of a real-world or financial asset, such as a bond, a fund share, or a unit of real estate, as a digital token that is transferred and settled directly on-chain, without the intermediaries a traditional financial system relies on. What began as experimentation has become institutional practice: in 2021 the European Investment Bank issued a 100 million euro bond on Ethereum [1], the Bank

Giovanni Pio Delvecchio, Stefano Fantazzini, Giacomo Frisoni, Gianluca Moro, Mario Bravetti, and Lorenzo Bacchiani are with the University of Bologna, Bologna, Italy.

Thomas Troels Hildebrandt is with the University of Copenhagen, Copenhagen, Denmark.

*These authors contributed equally to this work (co-first authorship).

Manuscript received September 28, 2026.

**The generated smart contract compiles.
Yet it does not behave correctly at run time.**

Input: the smart contract skeleton

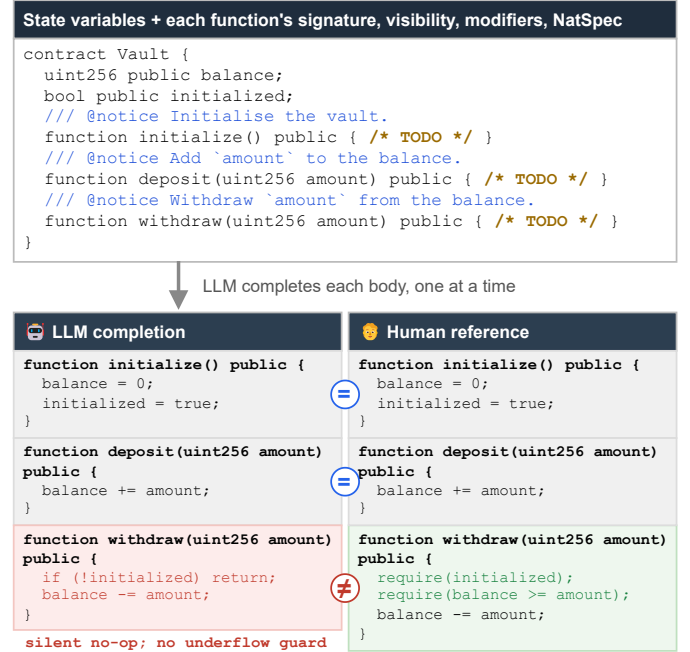


Fig. 1. **Compiling is not behaving.** Given a Vault contract with its function bodies removed (top), a model regenerates them one at a time (bottom left), shown against the original implementation (bottom right). initialize and deposit are regenerated correctly (=); withdraw is not (≠). Both versions compile, so compile-rate metrics cannot tell them apart; the behavioural oracles of our benchmark are built to catch exactly this divergence.

for International Settlements has since identified tokenisation as a structural shift in how financial markets operate [2], and in 2024 the chief executive of BlackRock, the world's largest asset manager, stated that "the next generation for markets will be the tokenisation of every financial asset" [3], [4]. Nor is finance the only domain: supply chains, for instance, rely on the same guarantees of transparency and auditability, and IBM's Food Trust platform cut Walmart's mango-traceability time from seven days to 2.2 seconds [5]–[7]. The contracts behind these systems collectively secure digital assets worth billions of dollars¹, yet, unlike conventional software, they cannot be patched once deployed. Any bug present in the

¹<https://defillama.com/>

bytecode at deployment is permanent and exploitable by anyone, so a single latent fault can cost real money, expose real users, and cannot be undone.

Large Language Models (LLMs) have demonstrated remarkable ability to generate syntactically valid Solidity code [8]–[10], and developers are increasingly integrating LLM-based code generation into smart contract workflows [8]. However, the dominant evaluation paradigm measures compilation success: whether the generated code compiles without errors. We argue that compilation is a necessary but insufficient filter: necessary because non-compiling code is unusable, but insufficient because syntactically correct Solidity can silently contain dangerous semantic faults that alter token ownership, destroy collected revenue, or enable financial attacks. Compilation says nothing about whether a contract behaves as intended, and it is precisely this behavioural gap that we target.

Since an arbitrary generated contract comes with no specification of correct behaviour other than the original it was regenerated from, the natural response is differential testing: deploy the generated contract and the original side by side on a sandboxed chain, submit the same calls to both, and flag any divergence in whether a call succeeds or reverts, in what it returns, in the events it emits, or in the storage state it leaves behind. What distinguishes differential-testing approaches is where the calls come from, and the literature has explored two sources. The first is historical transaction replay, adopted by SolContractEval [11]: the calls are the transactions that real users actually sent to the original contract on mainnet, replayed in their original order, so the test reflects how the contract was really used. The second is differential fuzzing, adopted by SolBench [10]: the calls are generated automatically by a fuzzer that searches the input space of each function for arguments on which the two implementations disagree, so the test covers inputs that no user ever sent. Both are sound and validated notions of behavioural correctness, and we adopt both. Yet neither can guarantee that the test reaches every state the contract is designed to enter. Historical replay preserves the order of real calls, but only of the calls that happened to occur: a state that no user ever brought about is never tested. Differential fuzzing explores each function’s inputs in depth, but tests functions in isolation, so a state that can only be reached by combining several calls is never tested either.

Figure 1 shows why this matters, using a toy Vault contract. The missing balance check in the regenerated `withdraw` is reachable only after `initialize` has been called; once the vault is initialised, an over-withdrawal is rejected by the original but executed by the regenerated code, which, in the absence of checked arithmetic, corrupts storage silently. Historical replay exposes the fault only if some recorded transaction happened to call `initialize` before an over-withdrawal, and fuzzing tests `withdraw` on its own. In both cases the fault goes unreported, not because the two implementations agree, but because the test never reaches the state in which they disagree.

Rather than replacing either approach, this paper unifies them within a single framework and contributes a third, complementary oracle. We model the contract’s intended lifecycle as a *Dynamic Condition Response (DCR) graph* [12], a

lightweight declarative formalism that captures the condition, response, exclusion, and inclusion relations among contract events. The graph then *steers* test-sequence generation toward the prerequisite orderings that matter, exercising behaviours that neither historical traffic nor function-level fuzzing reaches.

This work addresses two gaps: the absence of a behavioural benchmark for LLM-generated smart contracts, and the lack of oracles able to detect behavioural divergence from a contract’s intended functionality. Concretely:

- 1) **A family of three behavioural fidelity oracles (O1–O3)** that complement the standard `compile@k` pre-filter. O1 applies differential fuzzing, asking whether each regenerated function remains equivalent to the original under coverage-guided input exploration. O2 replays historical mainnet transactions, asking whether the regenerated contract reproduces the behaviour observed on-chain. O3 replays test sequences synthesised from an LLM-extracted DCR graph of the contract’s lifecycle, reaching states that historical traffic and unguided fuzzing miss.
- 2) **A neuro-symbolic transaction sequence generator** that combines the permissive declarative semantics of DCR graphs with on-chain guard evaluation and LLM-guided parameter synthesis. An LLM extracts the DCR graph from the contract’s source, and a deterministic graph-traversal algorithm enumerates legal event orderings and validates each call by executing it on the original contract, so correctness follows from the algorithm’s guarantees rather than the model’s output.
- 3) **An open behavioural benchmark and replication package** available at <https://github.com/disi-unibo-nlp/sol-beh-ave> (MIT license). It contains the 250 benchmark contracts with their deployment metadata, the DCR graph extracted for each contract, the transaction sequences, and the scripts to regenerate contracts and run the three metrics.

Applying the framework to seven state-of-the-art open-weight models yields three findings. First, compilation is a poor proxy for correctness: even after automated repair, the models compile between 26.4% and 92.7% of their contracts, but only a minority of the regenerated function bodies that compile are equivalent to the original under differential fuzzing (31.4%, pooled over models and samples), and for six of the seven models at least one regenerated contract in eight diverges from the original under each replay oracle. Second, the three oracles are complementary rather than redundant: each detects divergent contracts that the others accept as correct, because each exercises the contract differently. In particular, DCR-guided sequences call functions that real users never did, typically owner-only setup and administrative operations, on one contract in three; this holds regardless of which model extracts the graph. Third, LLM-extracted DCR graphs are reliable enough to drive the oracle: against human-annotated ground truth over four relation types (`condition`, `exclusion`, `inclusion`, `response`), extraction reaches 0.72 precision and 0.88 recall overall, and 0.96 precision on `condition`, the relation that governs call ordering.

Paper structure: Section II introduces the necessary blockchain and smart-contract background. Section III surveys

related work. Section IV formalises the framework and its oracles. Section V reports the empirical results. Section VI discusses threats to validity. Section VII concludes.

II. PRELIMINARIES

a) Blockchain: A shared ledger of transactions, replicated across many independent nodes, that grows only by appending blocks. Each block contains a cryptographic hash of the previous one, so altering any past transaction would invalidate every block after it. Once recorded, the history is therefore both permanent and publicly visible, which is what makes it possible to replay a contract’s entire past.

b) Ethereum: The blockchain platform this work targets [13]. It runs programs through the Ethereum Virtual Machine (EVM), where every operation consumes *gas*, the unit in which computation is priced.

c) Smart contract: A program stored on the blockchain together with its persistent data, its *storage*. Its code cannot be changed after deployment, it runs only when a transaction calls one of its public functions, and its execution is deterministic: the same call on the same storage always yields the same result, so any transaction, past (O2) or newly generated (O3), can be re-executed with identical effects. Only state-changing functions can alter storage; read-only *view* and *pure* functions cannot, and are therefore excluded from behavioural comparison, while the *constructor* runs once at deployment to set the initial storage.

d) Transaction, revert, and event: A transaction is a call to a contract function, with given arguments, from a given sender. It either *succeeds*, making its storage changes permanent, or *reverts*, undoing them, typically because a check such as *require* failed. While running, a contract may emit *events*, log entries through which applications outside the chain observe what it did. Success or revert, return value, emitted events, and resulting storage are the outcomes on which our oracles compare an original contract with a regenerated one.

e) Solidity: The most widely used language for EVM contracts and the language of the code we evaluate [14]. A contract declares state variables and functions, each with a visibility. A function may carry *modifiers*, reusable checks run before its body (for instance, restricting it to the owner), and NatSpec comments (*@notice*, *@param*, *@return*) documenting it in natural language. Compiling a contract produces, besides its bytecode, an Application Binary Interface (ABI): the list of its functions with their argument and return types, which tells a caller how to format a call to the contract and how to read the result.

f) Etherscan: A public website and API that indexes every Ethereum transaction and hosts the source code of contracts whose deployers submitted it, verified against the deployed bytecode [15]. We use it for both the past transactions (O2) and the verified source of our benchmark contracts.

g) Hardhat: A development environment that runs a private Ethereum node locally [16]. We use it to deploy the contracts under test on a clean chain and to execute the transactions of O2 and O3.

h) Echidna: A fuzzer for Solidity that generates large numbers of transactions, guided by the code they reach, in search of one that violates a user-specified property, such as two implementations returning different results [17]. It is the engine of our differential-fuzzing oracle (O1).

i) Tools appearing in related work: None of the following is part of our pipeline. *Slither* [18] analyses Solidity source without running it, looking for known vulnerability patterns. *Mythril* [19] explores a contract’s possible executions symbolically, with an SMT solver [20], to find inputs that trigger vulnerabilities. *Diffusc* [21] fuzzes two versions of the same contract, before and after an upgrade, in parallel with Echidna and reports behaviours on which they differ.

III. RELATED WORK

a) Evaluating LLM-generated contracts: Early evaluations of LLM-generated Solidity rely on static analysers such as Slither and Mythril [22]–[24]. These scale well, but examine the code without running it, so they cannot observe how the generated contract actually behaves. Execution-based evaluation came later, in the form of differential testing: the generated contract and its original are run on the same inputs and any discrepancy is flagged. This is also the foundation of our O1 (differential fuzzing) and O2 (historical replay). SolBench [10] applies differential fuzzing to 7,604 contracts, and SolEval [9] complements compile success and pass@k, measured against human-written unit tests, with gas consumption and vulnerability rate. Both draw their inputs at random or from fixed tests, so neither can guarantee that call sequences requiring specific preconditions are exercised. SolContractEval [11] instead replays real historical transactions against the generated contract. Replay preserves the original call ordering, but it is bounded by the transaction history itself. Existing benchmarks thus trade scale for behavioural coverage; Table I positions the closest ones along the dimensions that distinguish ours.

b) Testing hand-written contracts: A substantial body of work tests *hand-written* contracts through fuzzing [17], [25], symbolic execution [19], [26], static analysis [18], and formal verification [27]; LLM4FUZZ [28] additionally steers a fuzzer with correctness conditions inferred by an LLM. All of these tools check a contract against properties stated explicitly by the developer, such as an invariant, a vulnerability pattern, or a formal specification, and take the code itself as the intended program. For generated code, there is no such list of properties: the contract can deviate from the intended behavior in any way.

c) Sequence-aware fuzzing: Differential fuzzing, as used in SolBench and in our O1, varies the arguments of one function at a time. A separate line of bug-finding fuzzers for hand-written contracts varies both: each test is a sequence of several calls, and the fuzzer searches over the order of the calls as well as over their arguments, because some branches can only be reached after other functions have modified the state. IR-Fuzz [29] infers the order from the code: if a function reads a state variable that another function writes, the writer is scheduled first. MuFuzz [30] instead tracks which sequences bring execution close to an unexplored branch and mutates those, rather than random ones, to reach it. Both report

TABLE I
COMPARISON OF SOLIDITY CODE-GENERATION BENCHMARKS.

Benchmark	#Contracts	Contract source	Evaluation [♣]	Fresh deploy + reprod.-verified [♡]	Sequence awareness [♣]	Ordering-aware coverage [◇]
Khalid et al. [8]	102	GitHub	Compile@k, Unit-test	✗	✗	✗
SolEval [9]	28	GitHub	Pass@k, Compile@k, Gas, Vuln. rate	✗	✗	✗
SolContractEval [11]	124	Etherscan	Hist. trans. replay	✗	✓	✗
SolBench [10]	7,604	Etherscan	Differential fuzzing	✗	✗	✗
SolBehave (Ours)	250	Etherscan	Compile@k, Differential fuzzing, Hist. trans. replay, DCR replay	✓	✓	✓

[♣] **Gas**: average gas per call. **Vuln. rate**: fraction of generated contracts flagged by static analysers (e.g., Slither).

[♡] **Fresh deploy + reprod.-verified**: the original contract is redeployed from its source code onto a clean blockchain and confirmed to reproduce its own past on-chain behaviour, rather than being tested against a live fork of the chain.

[♣] **Sequence awareness**: functions are called in an order that respects preconditions (e.g., initialising → withdrawing), not in isolation.

[◇] **Ordering-aware coverage**: the evaluation deliberately reaches code paths that are unlocked only by a specific prior call sequence, which real transaction histories rarely exercise.

substantial branch-coverage gains over order-agnostic fuzzers, confirming that call ordering is a first-order factor in exploring a contract’s state. We share the observation that call order matters, but we use it for a different purpose. These fuzzers look for bugs in a single contract, so they want sequences that reach as much of its code as possible, and they obtain them by analysing the code itself. We compare two contracts, so we want sequences that the original contract accepts as legitimate uses; only then can a difference in behaviour be blamed on the generated contract. Such sequences come from a description of how the contract is meant to be used, its DCR graph, rather than from an analysis of what its code happens to do.

d) Neuro-symbolic program analysis: Our sequence generator is neuro-symbolic: a neural component, the LLM, produces an artefact that a symbolic component, the graph-traversal algorithm, then executes with formal guarantees. In the taxonomy of [31], what distinguishes such systems is what the LLM produces. Ours produces a set of constraints that are already implicit in the contract, formalised as the relations of a DCR graph, which is an instance of *rule mining*; by contrast, in program-synthesis systems such as LINC [32], the LLM writes an executable logic program that a solver then runs.

e) DCR graphs: Introduced by Hildebrandt and Mukkamala [12] as a declarative, event-based model of loosely constrained processes, DCR graphs have been used to capture smart contract design patterns [33] and, in HighGuard [34], to monitor contract executions at run time against a developer-written specification. We differ on both counts: our graph is extracted automatically from the source rather than written by hand, and it is used proactively to generate transactions rather than to judge them. To our knowledge, this is the first use of DCR graphs for test-sequence generation in the evaluation of LLM-generated code.

IV. METHODOLOGY

Our framework proceeds in two stages. First, an LLM regenerates each benchmark contract (§IV-B). Second, the regenerated samples are evaluated against the deployed original

through a syntactic filter, Compile@k, and three behavioural oracles, namely differential fuzzing (O1, §IV-C), historical transaction replay (O2, §IV-D), and DCR-guided synthetic replay (O3, §IV-E). Throughout, C^* denotes the original contract, taken as ground truth, and $\hat{C}$ a regenerated sample. Both replay oracles compare C^* and $\hat{C}$ on the same quantities, defined in §IV-F. We first describe the set of contracts to which we apply the framework (§IV-A).

A. Benchmark Construction

We build the benchmark in three steps: crawling candidate contracts, keeping those that reproduce their own on-chain history on a fresh deployment, and restricting the set to contracts of a size the generation task can handle.

a) Crawling: We crawl Ethereum mainnet from block 16,308,190 (2023-01-01) onward, sampling blocks in batches and ranking call targets by transaction frequency. A candidate is kept only if it exposes verified source on Etherscan, is not a proxy, and has accumulated at least 200 historical transactions. This yields 8,360 candidates.

b) Reproducibility check: Each candidate is compiled from its verified source and deployed fresh on a local Hardhat node; constructor arguments are taken from Etherscan’s record, decoded from the deployment transaction, or defaulted. We then replay the contract’s first 200 historical transactions against this fresh deployment and compute the fraction that behave as recorded. Contracts that reproduce less than 80% of their own recorded history are discarded, leaving 2,098.

c) Size filter: We count the non-trivial functions of each contract, excluding constructors and `view/pure` functions, and retain only contracts with between 1 and 20 of them. Figure 2 shows the effect of this filter at the crawled and the reproducibility-verified stage: contracts with more than 20 functions dominate both sets, and the 1–20 band isolates exactly the 250 contracts of our final benchmark. The bound keeps the generation task tractable while preserving a meaningful range of complexity (median 16 functions, $p_{75} = 18$); §VI reports the ablation on which it is based.

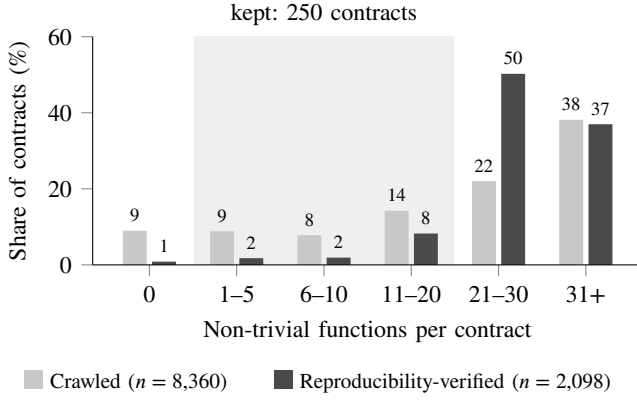


Fig. 2. **Distribution of non-trivial functions per contract at the two stages of benchmark construction.** Large contracts dominate both the crawled set and the subset that passes the reproducibility check; the size filter retains the shaded band, 1 to 20 functions, which contains the 250 contracts of the final benchmark (37, 40, and 173 in its three buckets).

B. Code Generation

Each of the 250 contracts is regenerated from its verified mainnet source by a stub-and-complete pipeline, following the protocol of SolBench [10], illustrated in Figure 1. All original comments are stripped and replaced by freshly generated NatSpec for every public function, so that no sample benefits from leaked documentation. The function bodies are then deleted, leaving the signature, visibility, modifiers, and NatSpec, and the model completes the stub one function at a time in source order. Later functions therefore condition on the code already generated for earlier ones: a function calling an internal helper sees that helper’s actual implementation rather than its stub. We repeat this procedure ten times per contract, independently and from the same skeleton, obtaining $k = 10$ samples that share every signature and differ only in their function bodies. The samples are compiled separately, and we report the fraction that compile, $\text{Compile}@k = c/k$ with c the number of compiling samples.

C. O1: Differential Fuzzing

O1 tests whether each regenerated function behaves as the original on arbitrary inputs. We consider a function *fuzzable* when it changes state and all its arguments are of primitive type. For every compiling sample and every fuzzable function f , we generate a harness contract that deploys C^* and $\hat{C}$ with identical constructor arguments and exposes a wrapper for f ; the wrapper invokes f on both deployments with the same arguments and asserts that the two calls agree on status and, when both succeed, on the returned data. Since the harness is itself a contract, it receives the return values, and since it deploys both contracts, it is their owner, so owner-only functions are exercised as well. Echidna [17] searches for arguments that violate the assertion (10,000 calls, sequence length 20, 120s timeout, following [10]); no LLM is used. A regenerated function is *equivalent* to the original if none is found, and O1 reports the fraction of equivalent functions. Each wrapper targets a single function, so O1 accumulates state across repeated calls to that function, but never interleaves different functions. Our numbers are not comparable

with those of SolBench [10], which fuzzes each generated function inside an otherwise original contract, so that no other function can be at fault. We fuzz inside a fully regenerated contract, as a developer would deploy it, where a fault in one function can corrupt the state that another one reads.

D. O2: Historical Transaction Replay

O2 replays each contract’s own recorded mainnet history against a freshly deployed instance. We take the first 1,000 historical transactions and deploy both C^* and $\hat{C}$ from source, never by injecting bytecode, which would leave the original deployer as owner and desynchronise every privileged call. The transactions are then re-issued against both contracts in their original order and compared along the axes of §IV-F. Nothing here is synthesised: the arguments and call order are taken from transactions submitted by real users, which is the strength of this oracle and, as §IV-E discusses, also its limit.

E. O3: DCR-Guided Synthetic Transaction Replay

Instead of depending on the orderings that real users happened to exercise, O3 extracts an explicit model of the contract’s intended call structure and synthesises sequences that reach the orderings and preconditions the historical record may never have covered.

1) *The DCR formalism:* A DCR graph [12] is a declarative process model that specifies which events may occur at each point and how the occurrence of one affects the others. We adopt the core formalism, since the extensions proposed since its introduction, such as time, data, and nesting, are not required for our purposes. DCR is deliberately *permissive*: it constrains only the relationships that must hold and leaves every other ordering free. This suits extraction by an LLM, which need only identify the few constraints that matter without over-constraining the process, and functions with no ordering requirement simply appear as isolated nodes, callable at any time. A graph consists of *events*, one per public state-changing function branch, each labelled with the function name and a caller role (*owner* or *any*). Every event is at all times either *included* or *excluded*: an included event can be executed and the constraints it places on other events apply, whereas an excluded event is ignored as if it were absent from the graph, so it cannot be executed, it no longer blocks the events it conditions, and any obligation on it lapses. The state of the graph, its *marking*, is therefore given by which events have been executed, which are currently included, and which are *pending*, that is, obliged to be executed but not yet. Events are related by four typed *relations*:

Condition, $A \rightarrow \bullet B$: if A is currently included, B is not enabled until A has been executed.

Response, $A \bullet \rightarrow B$: executing A makes B *pending*: B must be executed at some later point for the process to be complete, unless it is excluded first.

Inclusion, $A \rightarrow + B$: executing A includes B , so that B becomes executable again and its constraints and obligations apply once more.

Exclusion, $A \rightarrow \% B$: executing A excludes B , so that B can no longer be executed and the constraints and obligations attached to it are dropped.

We extend this core formalism in one respect. In DCR, a relation always applies; in a smart contract, whether it does can depend on the contract's state. We therefore allow a relation to carry a *guard*, a Solidity expression on the contract's state variables, and treat the relation as present only while the guard holds. One modelling convention is also worth noting. DCR has no primitive for an event that may be executed only once: this is modelled by an exclusion from the event to itself, which excludes it the moment it is executed. Since such self-exclusions are frequent and mechanical, our extraction schema records them as a flag on the event rather than asking the model to emit them as relations.

a) *Extraction:* The graph is extracted from the original contract, which is the specification the regenerated ones are judged against. We prompt an LLM with the contract's source and ABI and ask for the graph as JSON. The model emits one event per public state-changing function branch, and infers the relations between them, with their guards, from the preconditions that gate each function. For every event, it also proposes an argument value, later used to issue the call, and records the value of any compile-time constant a guard refers to, so that the guard can be evaluated during sequence generation. Because DCR under-constrains by design, the extractor emits few relations relative to the number of functions, 3.7 per graph on average across our benchmark. The prompts, the JSON schema, and the extracted graphs are available in the replication package. Figure 3 shows part of the graph extracted for GIFT0², selected to exhibit all four relation types. The contract keeps a record of how many units of a digital currency each address owns, and it also handled the initial sale of that currency: while the sale was open, anyone could send money to the contract and receive units in return, at a fixed price. The sale is controlled by the address that deployed the contract, its *owner*: `enableSale` and `disableSale` include and exclude `saleContinues`, the event of a successful purchase, so buying is possible only between the two. The money paid by buyers accumulates in the contract's own balance; `withdrawEth` transfers it to the owner's address, and only the owner can call it. A purchase therefore makes `withdrawEth` pending, under the guard `balance > 0`: there are proceeds to collect, unless the contract holds none. The condition from `setAllowance` to `transferFromSuccess` covers a separate mechanism of the token: holders can let another address spend part of their tokens, and the transfer is enabled only once that permission has been given. Finally, `transferFail`, the branch taken when a transfer is refused, is isolated: it can occur at any time and leaves everything else unchanged.

2) *Validated Sequence Generation:* Given a graph and a deployed instance of the original contract, we generate *validated sequences*: ordered lists of calls, each with concrete argument values, that respect the constraints of the graph and have all been executed successfully on a local Hardhat node.

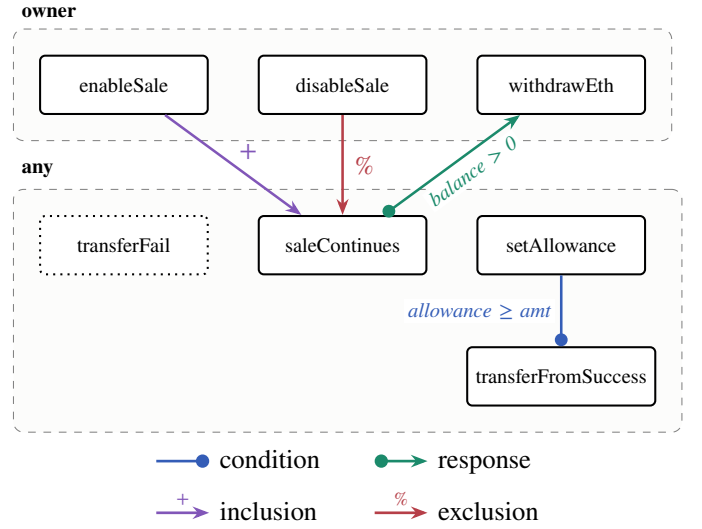


Fig. 3. Part of the DCR graph extracted for GIFT0. Dashed lanes group events by caller role; the dotted node is an isolated event; italic labels on relations are guards. The subgraph exhibits all four relation types.

Algorithm 1 Validated sequence generation for one contract

Require: DCR graph G of the original contract C^* ; budget T of successful calls

Ensure: set of validated sequences D

```

1:  $D \leftarrow \emptyset$ ;  $n \leftarrow 0$ 
2: while  $n < T$  do
3:   deploy  $C^*$  fresh; reset the marking of  $G$ ;  $\sigma \leftarrow \langle \rangle$ 
4:   loop
5:     evaluate the guards of  $G$  on the current state of  $C^*$ 
6:      $E \leftarrow$  events allowed by the marking of  $G$  under the
       relations whose guards hold
7:     remove ownership-transferring events from  $E$ 
8:     if  $E = \emptyset$  or  $n = T$  then
9:       break
10:    end if
11:     $e \leftarrow \text{SELECT}(E) \triangleright$  obligation, prerequisite, or least
       executed (exploration)
12:    execute  $e$  on  $C^*$  with its extracted arguments
13:    if the call succeeded then
14:      append  $e$  to  $\sigma$ ; update the marking of  $G$ ;  $n \leftarrow n + 1$ 
15:    end if
16:  end loop
17:  execute the ownership-transferring events; append suc-
    cesses to  $\sigma$ 
18:   $D \leftarrow D \cup \{\sigma\}$ 
19: end while
20: return  $D$ 

```

Algorithm 1 summarises the procedure, which we describe in the order in which it runs.

a) *Sequence construction:* The procedure collects calls until a budget T of successful ones is reached, building one sequence at a time. Each sequence starts from a fresh deployment of the contract and the initial marking of the graph, and is extended one call at a time, each time determining which events the graph currently allows and picking one of them;

²etherscan.io/token/0xC5bBaE...957b09d

when no event can be executed, the sequence ends and a new one is started from a fresh deployment.

b) *On-chain guard evaluation*: Which relations currently apply, and hence which events are allowed, depends on the guards, and a guard such as `balance > 0` depends on the contract’s state. Before each step, the procedure reads the value of every variable a guard refers to directly from the deployed contract, through the corresponding getter, and evaluates the guards on those values. This avoids having to predict how each call changes the state, and guarantees that the values used are the ones the contract really holds. Only events whose guards hold remain candidates for the selection described next.

c) *Event selection*: The event to execute is chosen among the allowed ones according to a fixed priority. Highest priority goes to *obligations*: events that a response from an already executed event requires to happen. Next come *prerequisites*: events that other events are waiting for. The remaining allowed events are chosen last, by an upper-confidence-bound rule over the attempts and successes recorded for each event during the run on that contract: events not yet attempted come first, then those with the highest success rate, with a bonus for events attempted rarely, so that no part of the contract is left unexercised. The first two priorities are applied with probability $\beta = 0.7$ and $\alpha = 0.8$ respectively, rather than deterministically, to preserve a degree of exploration beyond the constraints encoded in the graph. Both values were selected after preliminary tests as a balance between constraint-driven progress and exploration.

d) *Ownership-transferring events are executed last*: Some functions, such as `renounceOwnership` or `transferOwnership`, permanently take the owner role away from the address that deployed the contract, which is the address our procedure calls from. If such a function were executed in the middle of a sequence, every later call reserved to the owner would be rejected, and the rest of the sequence would be lost. These events are therefore excluded from selection during the T steps and executed at the very end, once the rest of the sequence has been built. Few contracts are affected: on the graphs extracted by GEMMA-4-31B, the strongest extractor in our experiments (§V), ownership-transferring events amount to 0.52 per contract on average, against 6.0 events per contract overall, and 71.6% of the contracts have none.

e) *Only admissible calls are retained*: The chosen event is executed on the original contract with the argument values that the LLM proposed for it when extracting the graph; if the call reverts, it is discarded and the marking is left unchanged, so a sequence contains only calls the original accepts. The DCR graph describes how the contract is meant to be used, and the sequences are meant to exercise exactly that: legitimate calls, in a legitimate order, whose outcome on the original is by definition the correct one. A call the original rejects does not belong to this intended use and is therefore left out; for a call the original accepts, any different outcome on the regenerated contract can only be the regenerated contract’s fault. Across the 250 contracts, Algorithm 1 attempted 256,923 calls and kept 212,904; the remaining 44,019 (17.1%) reverted and were discarded. The discard rate varies widely from contract to

contract: its median is 4.9%, but for 37 contracts it exceeds 98%, and for 32 of them it reaches 100%, leaving those contracts with no sequence at all. The cause is the same in every case: since we do not search for new arguments when a guard fails (0.9% of the kept calls fall back on arguments decoded from historical transactions, the rest use the LLM’s), a contract whose guards the extracted arguments can never satisfy loses most or all of its calls. Pairing the orderings of the DCR graph with a fuzzer for the arguments is the clearest direction for future work.

F. Comparison Axes

All three oracles decide divergence on the same criterion, the *status* of a call: whether it succeeds on both deployments or reverts on both. O1 also compares the value returned by successful calls, which its harness receives as the calling contract (§IV-C). O2 and O3 cannot: they replay transactions sent by user accounts, and Ethereum does not deliver return values to user accounts. What a user receives is the transaction’s receipt, which records status, gas used, and emitted events, and it is this receipt that the replay oracles compare, with status as the criterion and gas and events as secondary observations. The rest of this section concerns O2 and O3, which differ only in where their transactions come from: each transaction is issued, in the same order and from the same sender, to C^* and to every sample $\hat{C}$, and its outcome on the sample is compared with its outcome on C^* call by call.

a) *Divergence criterion*: The fraction of calls on which a sample agrees with C^* on status is its *status match rate*, and a contract is *divergent* under an oracle if at least one of its samples disagrees with C^* on at least one call.

b) *Secondary observations*: For every call we also record the other fields of the receipt, released with the replication package and used to check the status results rather than as criteria. *Gas*: the gas used by $\hat{C}$ relative to C^* ; two implementations of the same logic legitimately differ by a few percent, so only deviations beyond $\pm 10\%$ are flagged, as they usually indicate different logic. *Events*: the number of events the call emits and, for each, its topics, that is, the event signature and indexed arguments, so that the same occurrences are reported with the same key values. In addition, we periodically read the values returned by the contract’s zero-argument `view` functions, as a snapshot of the state it exposes. On our benchmark, none of these observations exposed a divergence that status had not (§VI).

c) *Aggregation*: The status match rate of a contract is the mean over its k samples; we also keep the *minimum*, since a fault exposed by one sample can be masked by the mean over ten. O2 can be applied to every contract with at least one compiling sample, whereas O3 also requires at least one validated sequence, which exists for 218 of the 250 contracts (§IV-E2); §V therefore reports, for each oracle and model, the number of contracts actually compared, which is the denominator of its divergence figures.

V. EVALUATION

We apply the framework to the 250-contract benchmark of §IV-A, regenerated by open-weight LLMs spanning several

families and a range of sizes, so that the findings are not tied to a single model or vendor. After the experimental setup (§V-A), we report compilation rates (§V-B) and the results of differential fuzzing (§V-F); we then assess the quality of the extracted DCR graphs, on which O3 depends (§V-C), compare the two replay oracles (§V-D), and explain their relationship through the functions each of them reaches (§V-E).

A. Experimental Setup

a) *Models*: We evaluate seven models from five families: GEMMA-4-31B and GEMMA-4-12B (Google, instruction-tuned), QWEN3.6-27B and QWEN3-CODER-30B (Alibaba; the latter a mixture-of-experts model with 3B active parameters, specialised for code), ORNITH-35B, CODESTRAL-22B and DEVSTRAL-24B (Mistral, both specialised for code). All run as 4-bit GGUF quantisations served by llama.cpp on a single RTX 5090, with a 60k-token context; generation uses temperature 0.7 and at most 2,048 output tokens per function. Each model plays two roles: it regenerates every benchmark contract with $k = 10$ samples (§IV-B) and it extracts the DCR graph of every original contract (§IV-E1).

b) *Repair*: A regenerated sample that fails to compile is given one repair attempt by the same model that produced it: the model receives the sample and the compiler error, with the instruction to fix only the reported error, at temperature 0.2. The protocol is identical for every model; samples that still fail to compile are excluded from behavioural evaluation.

c) *Oracles*: Against every compiling sample we run O1 with the configuration of §IV-C, and O2 and O3 with the same budget of 1,000 transactions per contract: the first ones recorded on chain for O2, and the successful calls collected by Algorithm 1 with T equal to that budget for O3. Since every model extracts graphs, O3 can be run either with the graphs of the model under test or with those of a single *reference extractor*; which model plays this role is decided on the results, in §V-D. All runs use seed 42.

B. Compilation Separates Models but Does Not Certify Them

Table II reports Compile@10 before and after repair. Raw rates span 18.8% to 89.1% and remain widely separated after repair (26.4% to 92.7%). Repair lifts every model by 3.6 to 10.8 percentage points but *preserves the ranking exactly*, and the relative gain is largest for the weakest models (+40% for DEVSTRAL-24B, +4% for GEMMA-4-31B). Whether the code that compiles is also correct is the question the remaining subsections answer.

C. Extracted DCR Graphs Are Reliable

O3 is only as good as the graphs it runs on, so before comparing the oracles we validated a sample of them against human judgement. We validated the graphs of GEMMA-4-31B, the strongest generator (Table II), on 30 contracts, drawn with a fixed seed by stratified sampling over three size tiers (1–5, 6–12, and 13–20 non-trivial functions) and constrained to include every relation type, since *response* is about 7% of extracted relations and a plain random draw could miss it. A single annotator, an author of this paper and a co-designer of the DCR

TABLE II
COMPILE@10 BEFORE AND AFTER ONE REPAIR PASS, PER MODEL.
Raw: SHARE OF THE TEN SAMPLES PER CONTRACT THAT COMPILE AS GENERATED; *Repaired*: AFTER A SINGLE ATTEMPT IN WHICH THE SAME MODEL FIXES ITS NON-COMPILING SAMPLES FROM THE COMPILER ERROR;
Gain: THEIR DIFFERENCE IN PERCENTAGE POINTS.

Model	Compile@10		
	Raw	Repaired	Gain
Gemma-4-31B	89.1%	92.7%	+3.6
Qwen3.6-27B	64.7%	71.3%	+6.6
Qwen3-Coder-30B	53.2%	61.8%	+8.6
Gemma-4-12B	48.0%	57.9%	+9.8
Ornith-35B	41.2%	52.1%	+10.8
Codestral-22B	34.3%	39.9%	+5.6
Devstral-24B	18.8%	26.4%	+7.6

TABLE III
HUMAN VALIDATION OF THE DCR RELATIONS EXTRACTED BY GEMMA-4-31B ON 30 CONTRACTS. AN EXTRACTED RELATION IS A TRUE POSITIVE (TP) IF THE ANNOTATOR CONFIRMS IT IN THE CONTRACT'S SOURCE AND A FALSE POSITIVE (FP) OTHERWISE; A FALSE NEGATIVE (FN) IS A RELATION THE ANNOTATOR FINDS IN THE SOURCE THAT THE EXTRACTOR DID NOT EMIT.

Relation	judgements			P	R	F1
	TP	FP	FN			
condition $A \rightarrow \bullet B$	65	3	12	0.96	0.84	0.90
response $A \bullet \rightarrow B$	18	6	1	0.75	0.95	0.84
inclusion $A \rightarrow + B$	21	23	0	0.48	1.00	0.65
exclusion $A \rightarrow \% B$	39	25	6	0.61	0.87	0.72
all	143	57	19	0.72	0.88	0.79

formalism, read the Solidity source before seeing the graph, judged every extracted relation as *correct*, *incorrect*, or *unsure*, and logged the relations the extractor omitted. Precision and recall are computed per relation type from these counts; sample, judgements, and sampling script are in the replication package.

Table III reports the results over 200 judged relations, none marked *unsure*: 0.72 precision and 0.88 recall overall. *Condition*, the relation that governs ordering and therefore drives sequence generation, is the most reliable (0.96 precision), and *response* is close behind (0.75 precision, 0.95 recall). *Inclusion* is the weakest (0.48), with a systematic error: a one-time precondition, such as an allowance that must first be set, is reported as if the target event had been excluded and were now being included again, when it was only not yet enabled. A second recurring error affects *exclusion* (0.61): a value driven to zero, such as an allowance fully spent, is reported as excluding a dependent event, although another function can restore the value. Both errors add relations rather than removing them, so they over-constrain the graph; the permissive semantics of DCR and the on-chain evaluation of guards absorb part of the effect, and §V-D shows that the residual differences between extractors do not alter the results.

D. The Two Replay Oracles Are Complementary

Figure 4 compares O2 and O3 in two settings. In the *matched* setting, each model's regenerated contracts are replayed against sequences generated from DCR graphs extracted by that same model; in the *fixed* setting, every model's contracts are replayed against the sequences of one reference

extractor, for which we take GEMMA-4-31B, the strongest generator (Table II) and, as §V-E shows, the extractor whose graphs reach the highest coverage. Panel (a) reports, for each oracle, how many contracts it compared, how closely they agreed with the original, and how many diverged; panel (b) splits the divergent contracts by the oracle that flags them.

a) *Neither oracle subsumes the other:* The decisive evidence is in panel (b). For every model, O3 flags contracts that O2 certifies as correct (the darkest segment, from 3 contracts for GEMMA-4-31B to 17 for CODESTRAL-22B), and for every model but GEMMA-4-31B the converse also holds (the lightest segment). For QWEN3.6-27B, for instance, of the 20 contracts flagged by O3, 13 are also flagged by O2 and 7 are not, while 16 contracts diverge under O2 only. The two oracles are therefore complementary instruments, not a stricter and a weaker version of the same test.

b) *The divergences track generation quality:* Two regularities indicate that the flagged divergences are real defects rather than noise. First, the worse a model is at producing compiling code, the more its compiling contracts diverge: under O2, the mean status match rate decreases at every step down the Compile@10 ranking, from 99.6% for GEMMA-4-31B to 88.2% for DEVSTRAL-24B, and under O3 it decreases along the same ranking, from 98.4% to 81.8%, with two adjacent models swapped. Second, the two oracles order the models almost identically (Spearman $\rho = 0.86$ between their status match rates).

c) *The result does not depend on the extractor:* O3 relies on an LLM-extracted graph, which raises the concern that a model could be flattered, or penalised, by the quality of its own graph. The fixed setting removes this degree of freedom: with every model judged by the reference extractor’s sequences, the number of contracts flagged by O3 only changes by at most two per model (from 3 to 17 with matched graphs, from 3 to 15 with fixed graphs; net -1 over the seven models) and remains non-zero for every model. The complementarity is a property of the two transaction sources, not of which model built the graph.

d) *The two oracles evaluate slightly different sets of contracts:* O2 can be applied to any contract with at least one compiling sample; O3 also needs a validated sequence for that contract (§IV-E2), which some contracts lack. The set of contracts O3 evaluates is therefore a subset of the set O2 evaluates, and panel (a) reports both sizes as N : 122 to 234 contracts for O2 and 85 to 174 for O3, depending on how many of the model’s samples compile. Hence, a contract flagged by O3 only was evaluated by both oracles, whereas a contract flagged by O2 only may be one that O3 never evaluated; §VI quantifies this.

e) *Cost:* On the runs of the reference extractor, O2 took 19.8 s per contract over its 234 contracts and O3 22.1 s over its 174. DCR extraction is a one-off cost per extractor, amortised over every model evaluated against its graphs.

E. Coverage Explains the Complementarity

The complementarity has a mechanical cause: an oracle can only expose a fault in a function it calls, and the two oracles

TABLE IV
FUNCTION COVERAGE OF O3 PER EXTRACTOR, AGAINST THE FIXED COVERAGE OF O2. *Coverage:* MEAN SHARE OF A CONTRACT’S STATE-CHANGING FUNCTIONS THAT O3 CALLS AT LEAST ONCE, WITH THE GRAPHS OF THAT EXTRACTOR (O2: 55.7% FOR EVERY ROW). *Contracts:* CONTRACTS ON WHICH O3 COVERS MORE FUNCTIONS THAN O2. LAST TWO COLUMNS: FUNCTIONS, OUT OF 1,759, THAT ONLY ONE OF THE TWO ORACLES CALLS.

Extractor	O3 coverage	Contracts O3 > O2	Functions covered by one oracle only	
			O3 only	O2 only
⚡ Gemma-4-31B	47.3%	84	311	511
🌀 Qwen3.6-27B	46.1%	81	305	520
⚡ Gemma-4-12B	44.6%	72	292	537
🔥 Devstral-24B	44.2%	77	301	535
🐦 Ornith-35B	41.6%	68	276	566
🔥 Codestral-22B	38.3%	60	263	596
🌀 Qwen3-Coder-30B	33.6%	35	246	619

do not call the same functions. We say that a state-changing function of a contract is *covered* by an oracle when at least one of the transactions the oracle replays calls it; state-changing functions are the only ones an oracle can exercise, and the benchmark has 1,759 of them. O2’s coverage is the same for every model, since it replays the same recorded transactions; O3’s depends on the graphs, hence on the extractor. Table IV reports, for each extractor, the mean per-contract coverage of O3, the number of contracts on which O3 covers more functions than O2, and how many functions each oracle covers that the other does not.

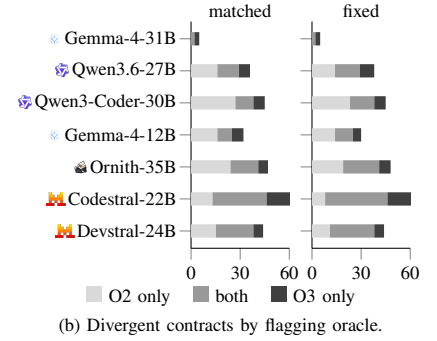
a) *Each oracle reaches functions the other does not:* O2 covers 55.7% of a contract’s state-changing functions on average, O3 between 33.6% and 47.3% depending on the extractor. The two figures are not nested. O3 reaches functions that no historical transaction ever called, between 246 with the weakest extractor and 311 with the strongest, one state-changing function in six; these are the functions O3 was built to reach. O2, in turn, reaches functions that O3 misses, between 619 and 511: functions whose guards the extracted arguments never satisfy, or whose relations the extractor did not emit. A better extractor shrinks the second set but leaves the first untouched, which is why the two oracles are used together.

b) *The mean hides a bimodal outcome:* The averages above conceal that the two oracles win on different contracts. With the graphs of GEMMA-4-31B, O3 covers more functions than O2 on 84 contracts, fewer on 121, and the same on 45; on the median contract O3 is ahead (57.1% against 50.0%), and only the mean favours O2. O3 wins on contracts whose functions are gated behind preconditions that real users never satisfied, which are exactly the contracts in which ordering-dependent faults hide; O2 wins on contracts with a wide, heavily used interface, the common case in a benchmark dominated by tokens, and these dominate the mean.

c) *Extracting DCR graphs and generating Solidity are different skills:* The models rank differently on the two tasks. By Compile@10, DEVSTRAL-24B is the weakest generator (26.4%); by the coverage its graphs give to O3, it is the third-best extractor (44.2%), ahead of ORNITH-35B, CODESTRAL-22B, and QWEN3-CODER-30B (33.6%). Only GEMMA-4-31B is first on both. Since a model’s graph does not reflect its

Model		 O2			 O3					
					matched			fixed		
		Compile@10	N	Status	$\hat{C} \neq C^*$	N	Status	$\hat{C} \neq C^*$	N	Status
 Gemma-4-31B	92.7%	234	99.6%	+2	174	98.4%	+5	174	98.4%	+5
 Qwen3.6-27B	71.3%	214	96.9%	+29	157	96.7%	+20	162	95.8%	+24
 Qwen3-Coder-30B	61.8%	183	95.5%	+38	125	96.1%	+18	136	95.5%	+22
 Gemma-4-12B	57.9%	169	95.4%	+25	127	94.6%	+16	129	94.6%	+16
 Ornith-35B	52.1%	193	94.4%	+41	140	96.1%	+23	147	95.2%	+29
 Codestral-22B	39.9%	159	91.1%	+46	115	81.8%	+50	123	81.9%	+53
 Devstral-24B	26.4%	122	88.2%	+38	85	85.5%	+29	89	83.3%	+33

(a) Per-oracle results.



(b) Divergent contracts by flagging oracle.

Fig. 4. **Compilation, historical replay (O2) and DCR-guided replay (O3) per model.** (a) N : contracts the oracle could compare; *Status*: mean share of replayed calls on which the regenerated contract $\hat{C}$ agrees with the original C^* on success or revert; $\hat{C} \neq C^*$: contracts with at least one disagreement, shaded in proportion to the value. O3 is run with the DCR graphs extracted by the model itself (*matched*) and with those extracted by GEMMA-4-31B for every model (*fixed*). (b) Each bar splits a model’s divergent contracts into those flagged by O2 only, by both oracles, and by O3 only, on matched and fixed graphs.

TABLE V

DIFFERENTIAL FUZZING (O1) PER MODEL, OVER THE $N=244$ CONTRACTS WITH A COMPILING SAMPLE AND A FUZZABLE FUNCTION. *Functions*: SHARE OF (FUNCTION, SAMPLE) PAIRS WHERE THE REGENERATED FUNCTION IS EQUIVALENT. *Contracts*: SHARE WITH A SAMPLE WHERE EVERY FUZZABLE FUNCTION IS EQUIVALENT.

Model	 O1: equivalent	
	Functions	Contracts
 Gemma-4-31B	33.4%	34.0%
 Qwen3.6-27B	35.5%	34.4%
 Qwen3-Coder-30B	26.6%	29.9%
 Gemma-4-12B	18.8%	13.1%
 Ornith-35B	16.9%	15.6%
 Codestral-22B	9.3%	5.3%
 Devstral-24B	2.0%	2.1%

generation ability, §V-D also judges every model with the same extractor’s graphs.

F. Differential Fuzzing Finds Few Equivalent Functions

O1 asks a stricter question than the replay oracles: not whether a regenerated contract accepts and rejects the same transactions as the original, but whether each of its functions returns the same result on any input Echidna can find. Of the 250 contracts, 244 expose at least one fuzzable function; the remaining six take only arguments of non-primitive type. O1 is run on every compiling sample of these 244 contracts, one harness per fuzzable function, and Table V reports its two measures per model.

a) *Most regenerated functions are not equivalent*: Averaged over all fuzzable (contract, function) pairs and all models, a regenerated function body is equivalent to the original in only 31.4% of the compiling samples, and only 34% of the contracts have at least one sample in which every fuzzable function is equivalent. The *Functions* column of Table V ranges from 2.0% for DEVSTRAL-24B to 35.5% for QWEN3.6-27B. The ranking closely tracks Compile@10 (Table II): five of the seven models keep their exact rank, and only the top two swap, with QWEN3.6-27B (35.5%) narrowly ahead of GEMMA-4-31B (33.4%) despite the latter’s higher compilation rate (92.7% against 71.3%). Even GEMMA-4-31B, whose samples compile most reliably, leaves two-thirds of its fuzzable functions non-equivalent to the original. The gap between compiling and

equivalent is therefore not a property of the weakest models but of all of them.

b) *Agreement with the replay oracles*: O1 flags a contract as soon as one of its functions is non-equivalent in one compiling sample, the same any-sample rule under which O2 and O3 flag a contract on a status disagreement (§IV-F), so the three divergent sets can be compared directly. Table VI does so over the contracts that all three oracles evaluated, with O3 run on the graphs of the reference extractor introduced in §V-D. O1 alone accounts for most of each model’s divergent contracts (38%–69% of N , the *O1* column of Table VI), far more than O2 or O3 alone (at most one contract, for any model). This is not a loss of signal from the replay oracles: O1’s per-function budget (up to 10,000 fuzzed calls) is far larger than O2/O3’s fixed 1,000-transaction budget, so within N it catches nearly every contract they flag — across all seven models, only one contract is flagged by a replay oracle without also being flagged by O1, consistent with a multi-transaction sequence dependency that single-function fuzzing cannot compose. The complementarity of §V-D therefore holds between O2 and O3, not between O1 and the replay oracles: within N , O1 is strictly the stronger test. All three oracles agree on 1.2%–30.8% of contracts per model (the *all* column), rising for the weaker models. O1’s advantage does not make O2 and O3 redundant: its inputs may be implausible, while O2/O3 divergences reflect realistic usage. The oracles differ structurally too: O1 cannot compose sequences, so it misses the state-dependent fault the one replay-only contract in Table VI represents.

VI. THREATS TO VALIDITY

a) *Internal validity*: O3 depends on an LLM-extracted graph: an incomplete graph lowers coverage, an over-constrained one blocks valid sequences. Three safeguards bound the effect. DCR’s permissive semantics make a missing relation cost coverage rather than correctness, since an unconstrained event is simply callable at any time; every guard is evaluated on the deployed contract, so a wrong guard can only prune a call, never admit one the original rejects; and the fixed-extractor experiment of §V-D replaces every model’s graphs by another model’s and moves the number of contracts flagged by O3 only by at most two per model, a perturbation far larger than any residual extraction error.

TABLE VI

DIVERGENT CONTRACTS BY THE COMBINATION OF ORACLES THAT FLAG THEM, PER MODEL. COUNTED OVER THE N CONTRACTS EVALUATED BY ALL THREE ORACLES, WITH O3 ON THE GRAPHS OF THE REFERENCE EXTRACTOR. A CONTRACT IS FLAGGED BY O1 IF AT LEAST ONE FUZZABLE FUNCTION IS NON-EQUIVALENT IN AT LEAST ONE COMPILING SAMPLE, AND BY O2 OR O3 IF AT LEAST ONE SAMPLE DISAGREES WITH THE ORIGINAL ON THE STATUS OF AT LEAST ONE TRANSACTION. THE SEVEN COLUMNS PARTITION THE CONTRACTS FLAGGED BY *at least one* ORACLE AND THEREFORE SUM TO AT MOST N ; THE REMAINDER AGREED WITH THE ORIGINAL UNDER EVERY ORACLE.

Model	N	one oracle only			two oracles			all
		O1	O2	O3	O1,O2	O1,O3	O2,O3	
Gemma-4-31B	170	93	0	0	0	3	0	2
Qwen3.6-27B	158	70	0	0	5	8	0	15
Qwen3-Coder-30B	133	50	0	0	15	7	0	14
Gemma-4-12B	126	87	0	0	10	5	0	11
Ornith-35B	143	99	0	0	7	6	0	21
Codestral-22B	120	59	0	0	7	15	0	37
Devstral-24B	85	46	1	0	4	6	0	26

The human validation of §V-C rests on a single annotator. Judging 200 relations and logging omissions requires reading the full source of 30 contracts with up to 20 functions each and tracing every `require` and branch, about 30 minutes per contract, or roughly 15 hours of expert time; a second annotator qualified to judge DCR semantics would have to be recruited from the handful of people who master the formalism, and the one we have is one of its designers. The judgement is also not a matter of taste: each relation is checked against a precondition that either is or is not in the source, and the two error patterns found (inclusion mistaken for condition, reversible zeroing mistaken for exclusion) are systematic, not idiosyncratic. The sample and the judgements are released, so the validation can be re-annotated independently.

Two design choices make our divergence counts lower bounds. We decide divergence on transaction status, not on the revert reason or on return values, which would require tracing each of the roughly 17 million replayed calls (250 contracts $\times$ 10 samples $\times$ 1,000 transactions $\times$ 7 models, per oracle); two samples that revert for different reasons therefore count as agreeing, so every divergence we report is real and some are missed. And O3 replays only calls that the original accepts, so it cannot catch a regenerated contract that accepts a call the original rejects; replaying the pruned calls as a negative test would attribute to the regenerated contract every failure of our own argument values, which is why we exclude it, but the 44,019 pruned calls are logged and the extension is mechanical.

Repair is performed by the model under test on its own output. A fixed external repairer would inject its own capability into every model’s score; self-repair keeps each model responsible for its own code, mirrors how a developer uses one assistant, and, as Table II shows, leaves the ranking of the models unchanged, so no conclusion depends on it.

b) External validity: The benchmark contains Etherscan-verified, non-proxy contracts with 1 to 20 non-trivial functions and is dominated by token-like contracts. That is the population of verified deployments, and the framework itself is contract-agnostic; whether the complementarity of the oracles

TABLE VII

COMPILE RATE AND GENERATION COST BY CONTRACT SIZE. *Mean* and *Range:* COMPILER@10 AFTER REPAIR, AVERAGED OVER THE SEVEN MODELS AND BETWEEN THE WORST AND THE BEST OF THEM. *s/sample:* SECONDS OF GENERATION PER COMPILING SAMPLE. THE DASHED LINE IS THE BENCHMARK’S 20-FUNCTION BOUND.

Functions per contract	Compile@10		s/sample
	Mean	Range	
1–5	+72.4%	54.2–88.9%	+4
6–10	+49.0%	35.0–69.4%	+13
11–15	+43.3%	24.2–64.2%	+24
16–20	+40.8%	16.7–70.6%	+41
21–25	+27.9%	3.9–46.4%	+75
26–30	+5.8%	5.6–7.2%	+502
31–35	+7.9%	2.8–14.4%	+415
36–40	+12.4%	2.8–22.8%	+310

holds on multi-step DeFi, bridges, or governance code, where ordering-dependent faults matter most, is the open question we consider most worth pursuing.

The 20-function bound is not a convenience but the result of an ablation (Table VII): we ran the full pipeline, repair included, on contracts binned by function count, 360 samples per bin and model. Compile@10 after repair decreases from 72.4% (1–5) to 40.8% (16–20), then drops to 27.9% at 21–25 and below 13% beyond, where repair stops compensating. Two effects make the cut necessary rather than merely convenient. Generation itself does not slow down with contract size, about one second per function body in every bin, but larger contracts need more attempts to yield a compiling sample, so the cost of one usable sample rises from 4 s (1–5) to 41 s (16–20) and 502 s (26–30): beyond the bound, most of the compute buys samples the oracles cannot evaluate. And past 25 functions the models stop being distinguishable: at 26–30, six of the seven compile exactly 20 of 360 samples and the seventh 26, whereas at 16–20 the strongest and weakest are still four times apart (70.6% against 16.7%). Extending the benchmark upward would therefore add contracts on which compilation is too rare to evaluate and too uniform to discriminate.

Fresh deployment starts every contract from its constructor state rather than from accumulated mainnet state, so a fault that manifests only late in a contract’s life can escape; in exchange, initialisation and access-control faults are detected reliably and the benchmark runs without an archival node. All models run as 4-bit quantisations with one temperature and $k = 10$ samples, identical for every model, so the comparison is fair even if absolute rates would shift under other settings; the selection probabilities α and β were not varied, but the fixed-extractor control, which perturbs the sequences far more than any plausible change of α or β , left the results unchanged.

c) Construct validity: We measure behavioural equivalence on the exercised transactions, not functional correctness. A regenerated contract that agrees with the original on every replayed call may still differ on inputs no oracle produced, which is precisely why O1 searches for distinguishing inputs instead of replaying a fixed set, and why the three oracles are used together. The original contract is our only specification, so a regenerated contract that fixes a fault of the original is scored as divergent; the admission filter of §IV-A, requiring each original to reproduce at least 80% of its own history,

guarantees that the reference is reproducible, not that it is correct, and any such case counts against the regenerated contract, in the conservative direction.

VII. CONCLUSION

A regenerated smart contract has no specification of correct behaviour other than the original it was regenerated from, so the original is the only available oracle. On this basis we built SolBehave, a benchmark of 250 reproducibility-verified mainnet contracts, and evaluated seven open-weight models with three behavioural oracles: differential fuzzing, historical transaction replay, and, our core contribution, replay of synthetic transactions generated from a DCR graph of the contract's intended lifecycle.

Three findings stand out. Compilation is a poor proxy for correctness: compile rates range from 26.4% to 92.7%, yet only 31.4% of the regenerated functions that compile are equivalent to the original, and for six of the seven models at least one compiling contract in eight diverges from the original under each replay oracle. The replay oracles are complementary rather than redundant: for every model, DCR-guided replay exposes divergent contracts that historical replay certifies as correct, and for all but one model the converse also holds, whichever model extracts the graphs; the reason is coverage, since DCR-guided sequences reach one state-changing function in six that no historical transaction ever called, typically owner-only setup and administrative operations. And LLM-extracted DCR graphs are reliable enough to drive the oracle, with 0.96 precision on the `condition` relations that govern ordering, while extracting a faithful graph and generating correct Solidity prove to be distinct capabilities.

Three directions follow directly. Extending the human validation to more contracts, extractors, and annotators would let extraction fidelity be correlated with oracle divergence per model. Extending the benchmark to DeFi protocols, bridges, and governance systems would test the complementarity where ordering-dependent faults matter most. And pairing DCR-guided replay with a fuzzer for the arguments, and with a negative-testing oracle that replays the pruned calls and checks that the regenerated contract rejects them too, would remove the two limits of the current corpus with no new machinery.

REFERENCES

- [1] European Investment Bank. (2021) EIB issues its first ever digital bond on a public blockchain. [Online]. Available: <https://www.eib.org/en/press/all/2021-141-european-investment-bank-eib-issues-its-first-ever-digital-bond-on-a-public-blockchain>
- [2] Bank for International Settlements, CPMI, "Tokenisation, money and the payment system," BIS, Tech. Rep., 2024. [Online]. Available: <https://www.bis.org/cpmi/publ/d225.htm>
- [3] L. Fink and R. Goldstein. (2024) Tokenization: Disrupting financial markets. [Online]. Available: <https://www.blackrock.com/corporate/literature/article-reprint/larry-fink-rob-goldstein-economist-op-ed-tokenization.pdf>
- [4] BlackRock. (2024) BlackRock launches its first tokenized fund, BUIDL, on the Ethereum network. [Online]. Available: <https://www.businesswire.com/news/home/20240320771318/en/BlackRock-Launches-Its-First-Tokenized-Fund-BUIDL-on-the-Ethereum-Network>
- [5] S. Ghosh, S. Alivea, M. S. Ali, S. Chatterjee, B. Banerjee, and I. Mukhopadhyay, "Tracking transparency: Blockchain in the global supply chain—a case study of ibm food trust," in *Advances in Computers*. Elsevier, 2026, vol. 142, pp. 555–564.
- [6] A. Mohammed, V. Potdar, and M. Quaddus, "Exploring factors and impact of blockchain technology in the food supply chains: An exploratory study," *Foods*, vol. 12, no. 10, p. 2052, 2023.
- [7] J. Joo and Y. Han, "An evidence of distributed trust in blockchain-based sustainable food supply chain," *Sustainability*, vol. 13, no. 19, p. 10980, 2021.
- [8] S. Khalid and C. Brown, "Evaluating capabilities and perspectives of generative ai tools in smart contract development," in *Proceedings of the 7th ACM International Symposium on Blockchain and Secure Critical Infrastructure*, ser. BSCI '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: <https://doi.org/10.1145/3709016.3737802>
- [9] Z. Peng, X. Yin, R. Qian, P. Lin, Y. Liu, H. Zhang, C. Ying, and Y. Luo, "Soleval: Benchmarking large language models for repository-level solidity smart contract generation," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Association for Computational Linguistics, 2025, pp. 4388–4411. [Online]. Available: <https://doi.org/10.18653/v1/2025.emnlp-main.218>
- [10] Z. Chen, H. Qin, N. Chen, X. Zhao, L. Xue, X. Luo, and X. Wu, "Solbench: A dataset and benchmark for evaluating functional correctness in solidity code completion and repair," *CoRR*, vol. abs/2503.01098, 2025. [Online]. Available: <https://doi.org/10.48550/arXiv.2503.01098>
- [11] Z. Ye, J. Chen, Z. Shao, L. Bao, X. Yang, and Z. Liu, "Solcontracteval: A benchmark for evaluating contract-level solidity code generation," in *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16-20, 2025*. IEEE, 2025, pp. 2858–2869. [Online]. Available: <https://doi.org/10.1109/ASE63991.2025.00235>
- [12] T. T. Hildebrandt and R. R. Mukkamala, "Declarative event-based workflow as distributed dynamic condition response graphs," in *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010*, ser. EPTCS, K. Honda and A. Mycroft, Eds., vol. 69, 2010, pp. 59–73. [Online]. Available: <https://doi.org/10.4204/EPTCS.69.5>
- [13] Ethereum Foundation. (2026) Ethereum development documentation. Accessed: 2026-08-03. [Online]. Available: <https://ethereum.org/devlopers/docs/>
- [14] Solidity Team. (2026) Solidity documentation, release 0.8.36. Accessed: 2026-08-03. [Online]. Available: <https://docs.soliditylang.org/en/v0.8.36/>
- [15] Etherscan. (2026) Etherscan developer APIs: Introduction. Accessed: 2026-08-03. [Online]. Available: <https://docs.etherscan.io/introduction>
- [16] Nomic Foundation. (2026) Hardhat documentation: Getting started. Accessed: 2026-08-03. [Online]. Available: <https://hardhat.org/docs/getting-started>
- [17] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 557–560. [Online]. Available: <https://doi.org/10.1145/3395363.3404366>
- [18] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 2019, pp. 8–15. [Online]. Available: <https://doi.org/10.1109/WETSEB.2019.00008>
- [19] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," *HITB SECCONF Amsterdam*, vol. 9, no. 54, pp. 4–17, 2018.
- [20] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.
- [21] Trail of Bits. (2024) diffusc: Differential fuzzing of upgradeable smart contract implementations. Accessed: 2026-08-03. [Online]. Available: <https://github.com/crytic/diffusc>
- [22] E. A. Napoli, F. Barbàra, V. Gatteschi, and C. Schifanella, "Leveraging large language models for automatic smart contract generation," in *48th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2024, Osaka, Japan, July 2-4, 2024*, H. Shahriar, H. Ohsaki, M. Sharmin, D. Towey, A. K. M. J. A. Majumder, Y. Hori, J. Yang, M. Takemoto, N. Sakib, R. Banno, and S. I. Ahamed, Eds. IEEE, 2024, pp. 701–710. [Online]. Available: <https://doi.org/10.1109/COMPSAC61105.2024.00100>

- [23] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, “Empirical review of automated analysis tools on 47, 587 ethereum smart contracts,” in *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 530–541. [Online]. Available: <https://doi.org/10.1145/3377811.3380364>
- [24] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, “Empirical evaluation of smart contract testing: what is the best choice?” in *ISSTA ’21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 566–579. [Online]. Available: <https://doi.org/10.1145/3460319.3464837>
- [25] C. Shou, S. Tan, and K. Sen, “Itfuzz: Snapshot-based fuzzer for smart contract,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 322–333. [Online]. Available: <https://doi.org/10.1145/3597926.3598059>
- [26] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 1186–1189. [Online]. Available: <https://doi.org/10.1109/ASE.2019.00133>
- [27] D. Jackson, C. Nandi, and M. Sagiv, “Certora technology white paper,” 2022.
- [28] C. Shou, J. Liu, D. Lu, and K. Sen, “Llm4fuzz: Guided fuzzing of smart contracts with large language models,” *CoRR*, vol. abs/2401.11108, 2024. [Online]. Available: <https://doi.org/10.48550/arXiv.2401.11108>
- [29] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, “Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting,” *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1237–1251, 2023.
- [30] P. Qian, H. Wu, Z. Du, T. Vural, D. Rong, Z. Cao, L. Zhang, Y. Wang, J. Chen, and Q. He, “Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing,” in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1972–1985.
- [31] G. P. Delvecchio, L. Molfetta, G. Moro *et al.*, “Neuro-symbolic artificial intelligence: A task-directed survey in the black-box models era,” in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, 2025, pp. 10 418–10 426.
- [32] T. Olausson, A. Gu, B. Lipkin, C. Zhang, A. Solar-Lezama, J. Tenenbaum, and R. Levy, “Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 5153–5176.
- [33] M. Eshghie, W. Ahrendt, C. Artho, T. T. Hildebrandt, and G. Schneider, “Capturing smart contract design with DCR graphs,” in *Software Engineering and Formal Methods - 21st International Conference, SEFM 2023, Eindhoven, The Netherlands, November 6-10, 2023, Proceedings*, ser. Lecture Notes in Computer Science, C. Ferreira and T. A. C. Willemse, Eds., vol. 14323. Springer, 2023, pp. 106–125. [Online]. Available: https://doi.org/10.1007/978-3-031-47115-5_7
- [34] M. Eshghie, C. Artho, H. Stammner, W. Ahrendt, T. T. Hildebrandt, and G. Schneider, “Highguard: Cross-chain business logic monitoring of smart contracts,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 2378–2381. [Online]. Available: <https://doi.org/10.1145/3691620.3695356>



Giovanni Pio Delvecchio received the bachelor’s degree in computer science from the University of Bari and the master’s degree in artificial intelligence from the University of Bologna, Italy. He is currently pursuing the PhD degree in computer science and engineering at the University of Bologna. His research interests include neuro-symbolic artificial intelligence, with a focus on its application to finance, law, and security.



Stefano Fantazzini received the bachelor’s degree in computer engineering and the master’s degree in artificial intelligence from the University of Bologna, Italy. He is currently pursuing the PhD degree in computer science and engineering at the University of Bologna. His research interests include automatic prompt learning, with a particular focus on soft prompt optimization, aiming to improve adaptation, efficiency, and reasoning across domains.



Giacomo Frisoni received the PhD degree in computer science and engineering from the University of Bologna, Italy. He is a postdoctoral researcher in the same institution. His research interests include language models and graph neural networks. He has co-authored more than 30 publications and received best-paper and reviewer awards.



Gianluca Moro received the PhD degree in computer engineering from the University of Bologna, Italy. He is an associate professor at the same institution, where he coordinates the NLP research group at the Cesena campus. He has co-authored 100+ publications. He serves on the editorial board of multiple international journals, and collaborates on scientific projects with research centers and companies.



Mario Bravetti is a Full Professor at the Department of Computer Science and Engineering of the University of Bologna. His research spans formal methods for distributed systems, concurrency, probability, and service-oriented and cloud computing. He has authored more than 100 publications, serves on editorial boards, and has held program roles at CONCUR, SEFM, and SAC.



Lorenzo Bacchiani received his PhD in computer science and engineering from the University of Bologna, Italy. He is currently a Research Fellow at the Department of Computer Science and Engineering, University of Bologna. His interests include behavioral methodologies for component interaction and adaptation, and digital twin technologies in the automotive domain.



Thomas Troels Hildebrandt received the PhD degree in semantics for concurrent systems from Aarhus University, Denmark. He is a professor in software engineering at the Department of Computer Science of the University of Copenhagen, where he heads the research section for Software, Data, People & Society. For more than ten years he has led interdisciplinary projects on reliable, people-centred software systems, including the DCR graph notation.