Engineering and Technology Journal e-ISSN: 2456-3358

Volume 11 Issue 05 May-2026, Page No.-10206-10216

DOI: 10.47191/etj/v11i05.60, I.F. – 8.595

The Effectiveness of Echidna in Detecting Reentrancy in Ethereum Smart Contracts Using Bug Injection

Muhammad Faruq[1], Rahmad Abdillah[2*], Nazruddin Safaat H.[3], Pizaini[4]

[1234] Department of Informatics Engineering, Faculty of Science and Technology, Universitas Islam Negeri Sultan Syarif Kasim Riau, Indonesia

ABSTRACT: Smart contract vulnerabilities, particularly reentrancy, have caused hundreds of millions of dollars in losses across the Ethereum ecosystem. While static analysis tools dominate current auditing practice, empirical evaluations have consistently demonstrated their high false negative and false positive rates for reentrancy detection. Dynamic analysis, exemplified by property-based fuzzing with Echidna, offers an alternative by evaluating contracts through actual execution. However, systematic empirical evaluation of dynamic tools under controlled ground-truth conditions remains limited. This study adapts the bug injection methodology, previously applied only to static analysis evaluation, to assess Echidna's effectiveness in detecting reentrancy. A dataset of 50 Solidity contracts was instrumented with oracle properties and injected with two reentrancy variants, single-function and cross-function, producing 100 ground-truth contract variants. Three fuzzing configurations of increasing intensity were evaluated across three metrics: detection rate, activation rate, and average detection time. Results show that Echidna achieved 100% activation but detected only 20% to 42% of injected bugs depending on the configuration and variant. Nearly all detections occurred within the first 25 seconds of each campaign, with no benefit from extended timeouts. These findings reveal a fundamental gap between bug reachability and exploitability confirmation under standard fuzzing conditions.

KEYWORDS: Blockchain Security, Echidna, Reentrancy, Bug Injection, Smart Contract Ethereum.

I. INTRODUCTION

Smart contracts are self-executing programs deployed on blockchain networks that automate agreement enforcement without relying on intermediaries. A defining characteristic of Ethereum smart contracts is their immutability: once deployed, the code cannot be modified or deleted [1], [2]. While this property guarantees execution transparency, it introduces a critical security consequence: any bug or vulnerability embedded in the deployed code becomes permanent and cannot be remediated through conventional software patching [3], [4]. This immutability stands in fundamental contrast to traditional web applications, where vulnerabilities discovered post-deployment can be addressed through updates.

The security risks of smart contracts have been empirically substantiated by alarming figures. In 2024 alone, over 150 documented incidents resulted in combined losses exceeding $328 million [5]. The OWASP Smart Contract Top 10 (2025) reported that 149 incidents caused total financial losses of $1.42 billion, averaging $9.5 million per incident [6]. Projections for 2025 suggest losses from smart contract vulnerabilities may surpass $3.5 billion [7]. Among the vulnerability classes responsible for these losses, reentrancy remains one of the most persistently exploited. In 2024, reentrancy attacks alone caused approximately $47 million in losses across 22 separate incidents [8].

The dominant approach to smart contract security auditing has been static analysis. However, empirical evaluations have consistently revealed its limitations. Using the SolidiFI bug injection framework, which systematically injected 9,369 bugs into 50 contracts, Ghaleb and Pattabiraman [9] found that all evaluated static analysis tools exhibited significant false negative rates, ranging from 129 to 4,137 missed bugs across seven vulnerability types. False positive rates were equally problematic: Slither reported 10.9%, Securify reached 25%, and Mythril performed worse [10]. Most critically, Huang et al. [11] demonstrated that more than 99.8% of reentrancy detections produced by leading static tools were false positives, effectively rendering their reentrancy warnings unreliable for auditors. Song et al. [22] further identified the root cause: existing detectors apply coarse detection rules that fail to recognize contracts protected by anti-reentrancy patterns, misclassifying them as vulnerable. These findings do not merely highlight tool deficiencies; they underscore the fundamental constraint of code-level inspection, which cannot observe runtime behaviors such as transaction ordering, conditional re-entry paths, and cross-contract interaction states that only manifest during execution [12], [13].

Dynamic analysis addresses this limitation by evaluating smart contracts through actual execution in a simulated blockchain environment [14]. Techniques such as property-based fuzzing, symbolic execution, and runtime tracing can

directly trigger exploit behavior and provide concrete proof-of-exploitability, rather than theoretical warnings [15], [16]. Echidna, a property-based fuzzer developed by Trail of Bits [17], exemplifies this approach: it automatically generates transaction sequences and reports violations of user-defined invariants, yielding counterexamples with high certainty of being true positives. However, despite these advantages, systematic evaluation of dynamic analysis tools, particularly against reentrancy, remains limited. Existing studies have measured code coverage or detection rates, but have not verified whether detected bugs are actually exploitable at runtime, leaving a critical gap in the empirical understanding of dynamic tool effectiveness [18].

To address this gap, this paper proposes a methodology that adapts the bug injection paradigm, previously applied only to static analysis tools, for the evaluation of dynamic analysis tools, with a focus on reentrancy detection by Echidna. Specifically, this study makes the following contributions: (1) an adaptation of the SolidiFI framework for dynamic analysis evaluation, incorporating oracle instrumentation via generic invariants to enable automated runtime verification; (2) a ground-truth dataset of 100 reentrancy-injected contract variants (single-function and cross-function) with verified compilability; and (3) an empirical evaluation of Echidna across three metrics, namely detection rate, activation rate, and average detection time per variant, providing the first objective benchmark for dynamic reentrancy detection under controlled ground-truth conditions.

II. BACKGROUND AND RELATED WORK

A. Reentrancy Vulnerability in Ethereum Smart Contracts

Reentrancy is among the most critical and historically consequential vulnerability classes in Ethereum smart contracts. It occurs when a function makes an external call to another contract before updating its own state variables, allowing the called contract, if malicious, to re-enter the original function recursively before the state is finalized [19], [20]. This pattern exploits a violation of the Check-Effects-Interactions (CEI) principle, which prescribes that all state updates must precede external calls [21]. The DAO exploit of 2016, which resulted in the loss of approximately $60 million in Ether, remains the canonical illustration of reentrancy's destructive potential [9].

B. Limitation of Static Analysis for Reentrancy Detection

Static analysis tools, including Slither [10], Mythril, Oyente, and Securify, operate by analyzing contract source code or bytecode without execution. These tools have become the dominant approach in smart contract security auditing due to their speed and scalability. However, their fundamental constraint lies in the inability to observe runtime behavior: execution order dependencies, conditional call paths, and inter-contract interactions that only emerge during actual execution cannot be reliably captured through code pattern matching alone [14].

This limitation has measurable consequences. The SolidiFI evaluation by Ghaleb and Pattabiraman [9] demonstrated that no static tool achieved adequate detection coverage, with false negative counts ranging from 129 to 4,137 bugs per tool across 9,369 injected bugs. A subsequent systematic review by Li et al. [23] confirmed persistent issues with taxonomic inconsistency and labeling ambiguity in static analysis benchmarks. Most critically for reentrancy specifically, Huang et al. [11] found that 99.8% of positive reentrancy detections by major tools were false positives, suggesting that static tools generate high-volume, low-reliability reentrancy warnings in practice.

C. *Dynamic Analysis with Echidna*

Dynamic analysis evaluates smart contracts through actual execution in a simulated or local blockchain environment, observing runtime behavior directly [14]. Key techniques include property-based fuzzing, symbolic execution, and runtime tracing [15]. Unlike static analysis, dynamic analysis can confirm exploitability: a bug that triggers a property violation during fuzzing has a high probability of being a true positive, as it requires an actual exploitable execution path [18].

Echidna [17] is a property-based fuzzer purpose-built for Solidity smart contracts. Developers define invariants as Solidity functions following the naming convention *echidna_*; Echidna then automatically generates and executes random transaction sequences, reporting any sequence that violates a defined invariant as a counterexample. Echidna employs both grammar-based and coverage-guided mutation fuzzing strategies to maximize state space exploration [17]. Its coverage guidance iteratively prioritizes transaction sequences that reach previously unexplored code branches, improving the probability of triggering complex vulnerability conditions.

D. Bug Injection as an Evaluation Methodology

Bug injection is a systematic evaluation technique in which known vulnerabilities are deliberately inserted into a codebase to measure the detection capability of analysis tools [24]. It overcomes a fundamental limitation of real-world contract datasets: vulnerabilities extracted from deployed contracts rely on manual labeling, cross-tool consensus, or post-exploit analysis, all of which introduce subjectivity, inconsistency, and potential mislabeling [23], [25]. Bug injection, by contrast, provides unambiguous ground-truth labels, since each injected bug has a known location, known type, and verifiable properties.

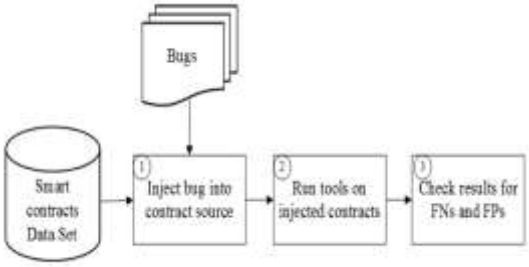

Figure 1 SolidiFI Workflow

SolidiFI [9] is the first framework designed specifically for systematic bug injection into Solidity contracts. Its pipeline consists of three stages: (1) injecting bugs from a predefined pool into contract source code via Abstract Syntax Tree (AST) parsing; (2) running static analysis tools on the injected variants; and (3) evaluating detection results to compute false negative and false positive rates. While SolidiFI established the methodology for static tool evaluation, it was not designed for dynamic analysis: it does not instrument contracts with oracle mechanisms for runtime verification, and its injected bugs are not validated for exploitability under fuzzing conditions.

E. Gap in the Literature and Research Position

Ren et al. [18] conducted the most directly comparable prior work, empirically evaluating dynamic testing tools using the SolidiFI dataset. Their study demonstrated that dynamic tools can achieve high code coverage and detect bugs missed by static tools. However, two significant limitations constrain their findings: (1) the SolidiFI dataset was designed for static tool evaluation and does not include oracle instrumentation, meaning the study measured only coverage-based detection without runtime exploitability verification; and (2) the study did not differentiate between bug detection and bug activation, a distinction critical to understanding dynamic analysis effectiveness. Specifically, activation rate, whether a fuzzer successfully executes the injected vulnerable code path, was not measured. Furthermore, the oracle problem in dynamic testing, the challenge of determining whether an observed runtime behavior constitutes a genuine exploit, was not addressed systematically [26].

The present study fills this gap by: (1) designing a bug injection dataset specifically for dynamic analysis, with automated oracle instrumentation embedded in each contract variant; (2) measuring both detection rate and activation rate as complementary metrics; and (3) evaluating Echidna, a modern, widely-used fuzzer not covered by Ren et al., against reentrancy variants under controlled ground-truth conditions.

III.METHODOLOGY

This study employs an experimental approach to evaluate the effectiveness of dynamic analysis in detecting reentrancy vulnerabilities through a controlled bug injection methodology.

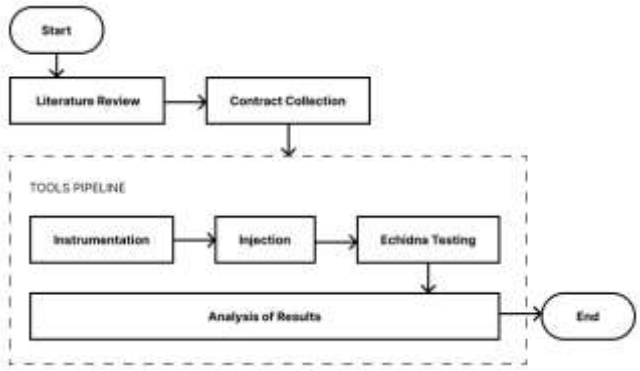

Figure 2 Methodology Flowchart

A. Literature Review

The literature review was conducted to establish a comprehensive understanding of reentrancy vulnerabilities, existing detection approaches, and gaps in the evaluation methodology for dynamic analysis tools. Literature was selected based on three criteria: publications addressing Ethereum smart contract security, particularly reentrancy; studies describing static or dynamic analysis approaches for vulnerability detection; and works employing bug injection methodologies or tool evaluation frameworks. From the collected literature, key information was extracted covering reentrancy definitions and technical characteristics, evaluation methodologies and datasets used, metrics applied, and limitations or gaps identified in prior work. This review informed the design of the bug injection dataset, oracle instrumentation strategy, and metric selection used in subsequent stages.

B. *Contract Collection*

A dataset of 50 public Solidity smart contracts was assembled from two primary sources: Etherscan and GitHub. Both sources were deliberately chosen to ensure the dataset reflects contracts that are publicly available, actively used or referenced in the smart contract security community, and structurally diverse enough to support meaningful reentrancy injection across a variety of fund-handling patterns.

Etherscan contributed contracts that are deployed and verified on the Ethereum mainnet and Sepolia testnet, representing real-world fund-holding contracts with external call patterns directly relevant to reentrancy. GitHub contributed the remaining contracts, drawn from open-source repositories [27] covering common fund-handling patterns including banking and deposit contracts, escrow arrangements, auction bidding pools, and reward distribution mechanisms.

All contracts were selected according to two technical criteria. First, each contract must contain at least one Ether transfer operation (using *transfer()*, *send()*, or low-level *call()*), as this is the prerequisite condition for a reentrancy injection point. Second, each contract must represent a fund-holding pattern commonly found in the DeFi ecosystem, such as vaults, crowdfunding pools, escrow arrangements,

auction bidding pools, or reward distribution mechanisms. Contracts that did not satisfy both criteria were excluded.

C. Instrumentation

Instrumentation is performed on each collected base contract prior to bug injection.

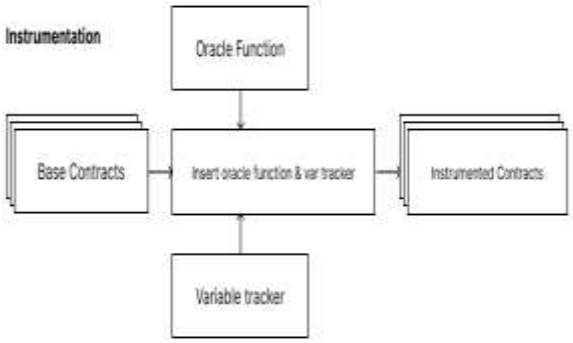

Figure 3 Instrumentation Workflow

As illustrated in Figure 3, each contract receives two additions: a pair of state-tracking variables and an Echidna oracle property function.

	Algorithm 1 *Run Instrumentation*
1	**procedure** RunInstrumentation
2	sol_files ← **all** .sol files **in** baseDir
3	**for each** file **in** sol_files
4	source ← ReadFile(file)
5	contract_name ← DetectContractName(source)
6	source ←InsertAfterContractOpen(source, contract_name, TRACKER_VARS)
7	source ← InsertBeforeContractClose(source, contract_name, ORACLE_FUNCTION)
8	WriteFile(outputDir / file, source)
9	**end for**
10	**return** results
11	**end procedure**

The tracking variables are inserted immediately after the contract's opening brace.

```
// Tracker variables (inserted after opening brace)
bool public  isReenteredHZ = false;
bool private lockedHZ      = false;
```

Figure 4 Tracker Variables

And the oracle property function is inserted immediately before the closing brace.

```
// Oracle function (inserted before closing brace)
function echidna_reentrantCheck() public view returns (bool) {
    return !isReenteredHZ;
}
```

Figure 5 Oracle Function

The oracle function follows Echidna's required naming convention and returns true under normal conditions. Once a reentrancy event is detected, the flag is set and the function returns false, triggering a property violation. This mechanism allows Echidna to treat reentrancy detection as a standard invariant-checking problem without any manual test case specification. The instrumented contract is then written to a separate output directory, preserving the original base contract unchanged.

D. Injection

Bug injection is performed on each instrumented contract that passed compilation verification.

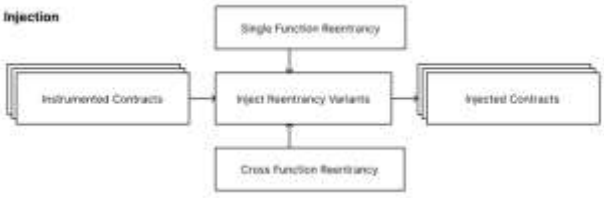

Figure 6 Injection Workflow

As illustrated in Figure 6, each valid contract is processed to produce two injected variants, one for each reentrancy pattern.

	Run Injection
1	**procedure** RunInjection(instrumentedDir, outputDir, variants)
2	sol_files ← all .sol files in instrumentedDir
3	**for each** file **in** sol_files **do**
4	source ← ReadFile(file)
5	contract_name ← DetectContractName(source)
6	mapping_var ← FindMappingVariable(source)
7	**if** mapping_var **not found then** mapping_var ← "dummyBalancesHZ"
8	**if not** HasReceiveFunction(source) **then** source ← InsertReceiveFunction(source, contract_name)
9	**for** each variant **in** variants **do**
10	source ← InjectVariant(source, mapping_var, contract_name, variant)
11	WriteFile(outputDir / file_variant.sol, source)
12	**end for**
13	**end for**
14	**end procedure**

The injector first identifies a mapping variable in the original contract to serve as the balance store for the injected bug; if none is found, a placeholder balance mapping is declared automatically to serve as the required state variable for the injected bug. If the contract does not already contain a *receive()* or *payable fallback()* function, one is injected automatically to allow the contract to receive Ether, a prerequisite for the fuzzer to trigger the reentrancy path. The

bug template is then inserted before the contract's closing brace, and the result is written as a separate file per variant.

	Inject Variant
1	**procedure** InjectVariant(source, mapping_var, contract_name, variant)
2	**if** mapping_var = "dummyBalancesHZ" then
3	dummy_mapping ← "mapping(address => uint256) public dummyBalancesHZ;"
4	**else**
5	dummy_mapping ← ""
6	**end if**
7	**if** variant = "single_function" **then**
8	code ← GenerateSingleFunctionTemplate(dummy_mapping, mapping_var)
9	source ← InsertBeforeContractClose(source, contract_name, code)
10	injection_line ← FindLine(source, "bug_reentrancy_single")
11	**else if** variant = "cross_function" **then**
12	code ← GenerateCrossFunctionTemplate(dummy_mapping, mapping_var)
13	source ← InsertBeforeContractClose(source, contract_name, code)
14	injection_line ← FindInjectionLine(source, "bug_reentrancy_cross_withdraw")
15	**end if**
16	**return** source, injection_line
17	**end procedure**

Two reentrancy variants are injected. The single-function variant injects one function that performs an external Ether transfer before updating the sender's balance, directly violating the Check-Effects-Interactions (CEI) principle.

```
// Variant 1: Single-function reentrancy
function bug_reentrancy_single(uint256 _amount) public {
    require(balances[msg.sender] >= _amount);
    if (lockedHZ) { isReenteredHZ = true; } // oracle trip
    lockedHZ = true;
    (bool ok,) = msg.sender.call{value: _amount}(""); // external call BEFORE state update
    unchecked { balances[msg.sender] -= _amount; } // state updated AFTER
    lockedHZ = false;
}
```

Figure 7 Single-Function Reentrancy Variant

The cross-function variant injects two cooperating functions: a withdrawal function that makes an external call before updating state, and a balance reader that returns the still-stale balance during reentrancy, enabling a more subtle multi-function exploit.

```
// Variant 2: Cross-function reentrancy
function bug_reentrancy_cross_withdraw(uint256 _amount) public {
    if (lockedHZ) { isReenteredHZ = true; } // oracle trip
    lockedHZ = true;
    (bool ok,) = msg.sender.call{value: _amount}(""); // external call first
    unchecked { balances[msg.sender] -= _amount; }
    lockedHZ = false;
}
function bug_reentrancy_cross_getBalance() public view returns (uint256) {
    return balances[msg.sender]; // reads stale balance mid-reentrancy
}
```

Figure 8 Cross-Function Reentrancy Variant

Both variants set an internal detection flag upon re-entry, which causes the oracle property to return false and trigger an Echidna violation report. The core structure of both injected bug templates is shown below.

E. *Echidna Testing*

Three Echidna experiment configurations were defined to evaluate how fuzzing intensity affects detection performance. Each configuration varies the core parameters controlling the number of test sequences, sequence length, shrink, and timeout. Table 1 summarizes the parameter differences across all three experiments.

Table 1 Echidna Experiment Configuration

Parameter	Light (Exp 1)	Medium (Exp 2)	Heavy (Exp 3)
testLimit	50,000	100,000	150,000
seqLen	100	150	200
shrinkLimit	5,000	10,000	15,000
timeout (s)	60	180	360
echidna_timeout (s)	75	195	375

The *testLimit* parameter defines the maximum number of transaction sequences Echidna will generate before concluding.

The *seqLen* parameter controls the number of individual function calls within a single generated sequence. Longer sequences allow Echidna to reach deeper contract states, which is especially relevant for cross-function reentrancy patterns that require multiple preparatory calls before the vulnerable path becomes reachable.

The *shrinkLimit* parameter governs how many simplification attempts Echidna may perform on a counterexample sequence once a property violation has been found. A higher shrink limit produces a more minimal and interpretable failing transaction sequence, which is beneficial when analyzing whether the oracle violation corresponds to a genuine reentrancy execution path.

The *timeout* parameter is the wall-clock time limit (in seconds) passed directly to Echidna's internal scheduler; it bounds the overall campaign duration regardless of whether *testLimit* has been reached.

An additional outer time limit is enforced by the pipeline's subprocess manager, set slightly higher than Echidna's internal timeout to allow Echidna to finalize its output before the process is terminated.

All remaining parameters (coverage, deployer, sender, balanceAddr, balanceContract, and maxTimeDelay) are held constant across all experiments to isolate the effect of intensity scaling on detection outcomes.

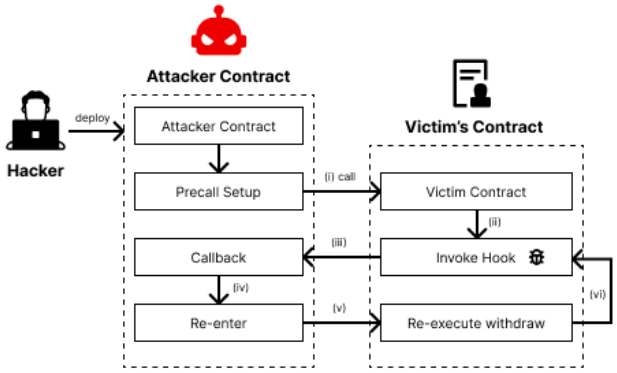

Figure 9 Attacker Contract Flowchart

To enable Echidna to trigger and detect reentrancy, the pipeline automatically generates an attacker contract wrapper for each injected contract variant. The attacker contract flow consists of six stages, as illustrated in Figure 9.

	Run Attacker Contract
1	**procedure** EchidnaAttacker(target)
2	isAttacking ← false
3	**function** attack(amount)
4	target.bug_reentrancy(amount)
5	**function** receive() **payable**
6	**if not** isAttacking **then**
7	isAttacking ← true
8	**try** target.bug_reentrancy(msg.value)
9	isAttacking ← false
10	**end if**
11	**end procedure**
12	**procedure** VictimEchidna **extends** Victim
13	attacker ← EchidnaAttacker(this)
14	**function** fuzz_attack(amount):
15	amount ← (amount **MOD** 10 ether) + 1
16	attacker.attack(amount)
17	**end procedure**

In the first stage (i), the Attacker Contract completes its precall setup and initiates the primary call to the Victim Contract.

In the second stage (ii), the Victim Contract processes the request and execution reaches the vulnerable external call, effectively moving to the Invoke Hook phase.

In the third stage (iii), the external call sends Ether back to the attacker, triggering a Callback to the Attacker Contract's fallback function.

In the fourth stage (iv), the Attacker Contract catches the callback and shifts to the Re-enter state.

In the fifth stage (v), the Attacker Contract immediately calls back into the Victim Contract to Re-execute withdraw before the victim's internal state is updated.

In the sixth stage (vi), the Victim Contract's execution loops back to the Invoke Hook with a still-stale balance.

Once the attack sequence completes, the pipeline processes Echidna's output to determine the detection result. If a property violation is confirmed, Echidna records the attack as a detected reentrancy and reports the minimal transaction sequence needed to reproduce the exploit via its shrinking phase. The pipeline then classifies the result as one of five outcomes: DETECTED, ACTIVATED, UNREACHABLE, TIMEOUT, or ERROR, along with the elapsed detection time in seconds.

F. *Analysis of Results*

The analysis stage processes raw Echidna output from all three experiment configurations and computes four evaluation metrics: detection rate, activation rate, average detection time per bug variant, and cumulative detection rate over time expressed as an Empirical Cumulative Distribution Function (ECDF). Each metric is computed per bug variant and aggregated as an overall score. In the comparative experiment setting, all metrics are also computed per configuration to evaluate the effect of fuzzing intensity on detection performance. Each Echidna result is first classified into one of five mutually exclusive status categories based on two binary signals derived from the pipeline output: whether the oracle property was violated and whether the injected bug's external call line was executed during fuzzing. Table 2 defines the classification scheme.

Table 2 Status Classification

Status	Condition	Meaning
Detected	property_broken=True	Oracle invariant triggered and return false
Activated	bug_line_hit=True property_broken=False	Bug patch reached but oracle not triggered
Unreachable	bug_line_hit=False property_broken=False	Fuzzer did not reach the injected bug code path
Timeout	Echidna timeout elapse before result	Detection time recorded as the maximum timeout value.
Error	Echidna process crashed or failed to execute	Excluded from rate calculation

Detection rate measures the proportion of injected bugs for which Echidna successfully violated the oracle property. This metric is adapted from Ghaleb and Pattabiraman [9] and computed as follows.

$$DR = \left(\frac{\text{Total DC}}{\text{Total IC}} \right) \times 100\%$$

Where Total DC is the count of contracts for which *property_broken* is true and Total IC is the total number of injected contract variants tested in that configuration.

Activation rate measures whether Echidna succeeded in reaching and executing the injected vulnerable code path, irrespective of whether the oracle property was ultimately violated. A contract is counted as activated when *bug_line_hit* is true. Activation is determined by inspecting Echidna's corpus coverage files: an asterisk marker on the external call line inside the injected bug function confirms that the fuzzer executed that line at least once during the campaign. The metric is computed as follows.

$$AR = \left(\frac{\text{Total AC}}{\text{Total IC}} \right) \times 100\%$$

Where Total AC is the count of contracts for which bug_line_hit is true and Total IC is the total number of injected variants tested.

Average detection time measures how quickly Echidna identifies a property violation. The timer starts when Echidna initializes the fuzzing campaign and stops when the first oracle property violation is reported. Contracts that are not detected within the configured time budget, including those terminated by timeout, are excluded. The metric is formulated as follows.

$$AT_v = \frac{\Sigma\, t_i}{\text{Total } DC_v}$$

Where AT_v is the mean detection time for variant v in seconds, t_i is the detection time of the i-th detected contract, and DC_v is the total count of contracts with *DETECTED* status for that variant. Average detection time is computed per variant and per experiment configuration.

In addition to the three primary metrics, this study visualizes the temporal distribution of detections using an ECDF over the fuzzing campaign duration. The ECDF captures what proportion of injected bugs have been detected by each elapsed second, providing a time-resolved view of detection performance that the scalar detection rate alone cannot convey. The function is defined as follows.

$$F(t) = \frac{1}{\text{Total IC}} \sum_{i=1}^{Total\ IC} 1(t_i \leq t)$$

Where t is the elapsed fuzzing time in seconds, t_i is the detection time of the i-th detected contract, Total IC is the total number of injected contract variants tested, and $1(t_i \leq t)$ is an indicator function that equals 1 if contract i was detected before time t, and 0 otherwise.

IV. RESULTS

A. *Detection Rate*

On the Single Function variant, detection rate increases consistently with configuration intensity. Exp 1 (Light) detected 28% of injected bugs, Exp 2 (Medium) detected 36%, and Exp 3 (Heavy) reached 40%. This monotonic progression confirms that a higher test limit and longer sequences contribute directly to improved detection for single-function reentrancy.

The Cross Function variant tells a different story. Light detected only 20% of injected bugs, while Medium achieved the highest detection rate across the entire experiment at 42%. Heavy, despite its larger configuration budget, fell back to 36%. This non-monotonic behavior suggests that beyond a certain configuration threshold, increasing fuzzing intensity does not guarantee better detection for bugs that require a coordinated sequence across two separate functions.

Figure 10 Detection Rate

Overall, detection rates remain low across all conditions, ranging from 20% to 42%, despite every bug having been successfully activated.

B. Activation Rate

Every experiment, across both variants and all three configuration intensities, produced a 100% activation rate. Echidna reached the injected bug code in every tested contract without exception.

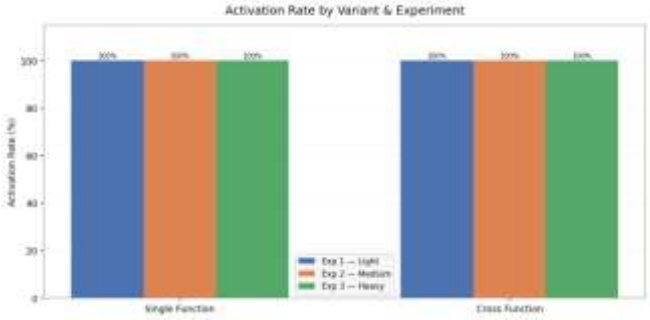

Figure 11 Activation Rate

This result validates the injection pipeline and the attacker wrapper generation in Steps 1 through 3, confirming that the instrumentation methodology operates independently of fuzzing configuration.

C. Average Detection Time

Detection time grows with configuration intensity across both variants. On the Single Function variant, Light detected bugs in an average of 10.9 seconds, Medium in 13.5 seconds, and Heavy in 17.0 seconds. On the Cross Function variant, Light averaged 8.2 seconds, Medium 14.5 seconds, and Heavy 19.5 seconds.

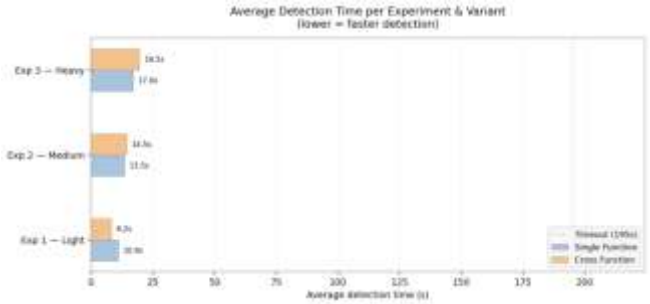

Figure 12 Average Detection Time

This pattern appears counterintuitive at first, but reflects a straightforward dynamic: heavier configurations explore a broader state space before revisiting the sequences that trigger reentrancy. Bugs detected under Light are, by nature, the easiest ones Echidna finds almost immediately, which pulls the average down. As configuration intensity grows, those same easy bugs are still found quickly, but the sample also includes harder bugs found later, raising the mean.

D. *Single Function Cumulative Detection Rate over Time*

All three experiments show the same structural pattern: detection is entirely concentrated within the first 25 seconds, after which every curve flattens completely and remains flat until the 195-second timeout. The final ordering matches the detection rate chart, with Heavy at 40%, Medium at 36%, and Light at 28%.

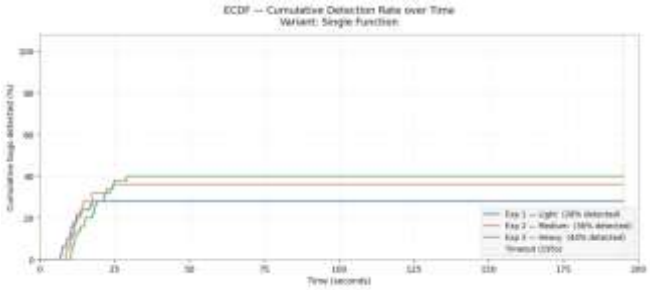

Figure 13 Cumulative Detection Rate over Time (Single Function)

The flat plateau spanning the remaining 170 seconds of each run carries a clear implication: additional fuzzing time beyond the initial burst produces zero additional detections for this variant. Echidna either finds a Single Function reentrancy bug in the early phase, or it does not find it at all.

E. Cross Function Cumulative Detection Rate over Time

The Cross Function ECDF reveals more nuanced behavior than its Single Function counterpart. Medium exhibits a two-phase detection pattern: a steep initial rise reaching approximately 40% by the 25-second mark, followed by a secondary step around second 45 that brings the final total to 42%. This secondary rise suggests that Medium's sequence length of 150 provides just enough capacity for Echidna to construct the specific multi-function call ordering required, after an initial exploration phase.

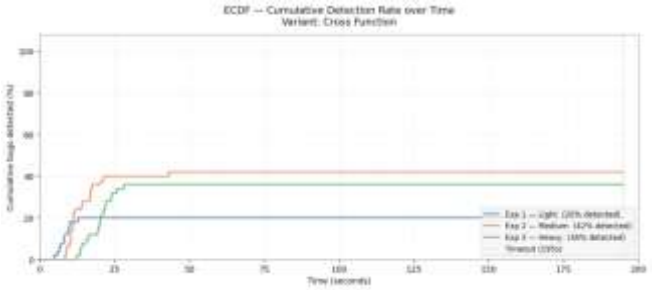

Figure 14 Cumulative Detection Rate over Time (Cross Function)

Heavy, despite its larger budget, plateaus at 36% without producing a second phase, while Light flattens earliest at 20%. Notably, all three curves reach their final values well before the 195-second timeout, reinforcing the finding from the Single Function analysis: for reentrancy detection, the critical window is short, and extended fuzzing campaigns offer diminishing returns regardless of variant type.

V. DISCUSSIONS

A. Detection Rate in Relation to Prior Static Analysis Benchmarks

The detection rates observed in this study, ranging from 20% to 42% across all configurations and variants, must be interpreted within the context established by prior evaluations of smart contract analysis tools. Ghaleb and Pattabiraman [9] reported that static analysis tools evaluated under SolidiFI exhibited false negative rates of 129 to 4,137 undetected bugs per tool across 9,369 injected instances. Scaled proportionally, even the best-performing static tool in that study, Slither, failed to detect a substantial share of reentrancy bugs, particularly those whose code structure deviated slightly from expected XPath patterns. In direct comparison, Echidna's 20–42% detection rate for reentrancy under the present study is numerically similar to or higher than the detection rates reported for static tools on the same vulnerability class. However, the two evaluations are not symmetrical: Ghaleb and Pattabiraman evaluated detection across seven vulnerability types on 50 contracts with 9,369 distinct injected bugs, while the present study targets reentrancy exclusively on 50 contracts with 100 variants. This narrower scope means Echidna's results are more precisely bounded but less generalizable across vulnerability types.

Ren et al. [18] provide the most structurally comparable reference point, as they also evaluated dynamic tools using the SolidiFI dataset. Their study found that dynamic tools could detect bugs that static tools missed, particularly through coverage-guided exploration, but coverage gains did not reliably correlate with vulnerability discovery. The present findings align with this observation: 100% activation confirms that Echidna reaches every injected bug's code path, yet detection remains bounded at 42%, demonstrating that code reachability and exploit confirmation are distinct problems. This distinction was not systematically measured

by Ren et al., which constitutes a methodological gap this study addresses.

B. The Activation–Detection Gap as a Structural Finding

The most consequential finding of this study is the persistent gap between activation rate (100%) and detection rate (20–42%). This gap represents a fundamental characteristic of property-based fuzzing applied to reentrancy: Echidna can consistently navigate to the vulnerable function, but the precise transaction sequence required to trigger re-entry before state is updated is generated only in a subset of cases.

This result has no direct parallel in the Ghaleb and Pattabiraman [9] study, which evaluated only static tools. Static analysis does not execute contracts, so the concept of activation, whether the analysis engine reaches and executes a vulnerable code path, is not applicable. All bugs that static tools fail to detect are simply missed during pattern matching or symbolic trace enumeration, not because the vulnerable path is unreachable.

Compared to Ren et al. [18], who found that dynamic tools achieved high code coverage but could not always confirm exploitability, the present study provides a more granular account of what "high coverage" actually means for reentrancy. Activation confirms execution of the vulnerable line; detection confirms a full exploit sequence with state violation. The 58–80 percentage point gap between these two metrics across all conditions demonstrates that coverage-based metrics are insufficient proxies for detection effectiveness, a finding consistent with Ren et al.'s observation but grounded here in a controlled ground-truth experiment rather than a coverage curve analysis.

C. Configuration Intensity and Variant-Dependent Behavior

The effect of configuration intensity on detection rate differs substantially between the two reentrancy variants. Single-function reentrancy responds monotonically: Light (28%), Medium (36%), Heavy (40%). This behavior is expected, as a single-function exploit requires only that Echidna constructs a transaction sequence in which a withdrawal call is re-entered before the balance update, a pattern that becomes increasingly likely as the test limit and sequence length grow.

Cross-function reentrancy exhibits a non-monotonic response: Light (20%), Medium (42%), Heavy (36%). The decline from Medium to Heavy is particularly notable. This pattern suggests that beyond a certain sequence length, the additional state space introduced by longer transaction sequences actively disrupts the coordinated call ordering required for cross-function exploitation. Ren et al. [18] similarly found that performance was not monotonic with depth limit for symbolic executors: beyond a depth of 50, Mythril's detection rate declined substantially. The present study identifies an analogous phenomenon for fuzzing-based tools, where the relevant parameter is sequence length rather than recursion depth. This parallel is meaningful: both findings indicate that parameter scaling has a point of diminishing or negative returns that is vulnerability-structure-specific, a nuance absent from both comparison papers.

D. Temporal Distribution: Early Detection and Diminishing Returns

The temporal analysis reveals that all detections, regardless of variant or configuration, occur within the first 25 seconds of each campaign. The remaining 170 seconds produce no additional detections. This finding contrasts sharply with the conventional assumption that longer fuzzing campaigns improve detection outcomes.

Ren et al. [18] demonstrated a related but distinct phenomenon: for dynamic tools, extending the execution timeout beyond a threshold increased the total number of bugs reported by static tools but did not change tool rankings. Their analysis, however, focused on timeout values ranging from seconds to hours across all vulnerability types, without examining the within-campaign temporal distribution for individual bug types.

E. Comparison of Methodological Strengths and Limitations Relative to Prior Work

Relative to Ghaleb and Pattabiraman [9]: The present study extends the SolidiFI bug injection paradigm to dynamic analysis by adding oracle instrumentation, a component that SolidiFI does not provide. This extension is necessary because dynamic tools require runtime property specifications to confirm exploitability; without oracle properties, a fuzzer can execute the vulnerable code path without producing a detectable signal. The trade-off is scope: Ghaleb and Pattabiraman evaluated six tools across seven vulnerability types with 9,369 injected bugs, while the present study evaluates one tool against one vulnerability type with 100 variants. The present study therefore provides deeper per-tool, per-variant characterization at the cost of breadth.

Relative to Ren et al. [18]: The present study narrows the scope of Ren et al.'s multi-tool empirical study to a single tool and single vulnerability class, enabling metric-level precision that their broader study could not achieve. Specifically, activation rate as a distinct metric from detection rate, and the per-second ECDF of detection timing, are absent from their analysis. However, Ren et al.'s inclusion of multiple tools (ContractFuzzer, sFuzz, ILF) and multiple datasets provides a cross-tool comparison dimension that the present study lacks. The two studies are therefore complementary: Ren et al. establish the landscape of dynamic tool performance, while the present study characterizes one tool's behavior with greater precision under controlled ground-truth conditions.

VI. CONCLUSIONS

This study evaluated Echidna's effectiveness in detecting reentrancy vulnerabilities using a bug injection methodology adapted from SolidiFI, across three configuration intensities

and two reentrancy variants on 50 instrumented Solidity contracts.

The central finding is a consistent gap between activation and detection. Echidna achieved 100% activation across all conditions, confirming that the instrumentation and attacker wrapper pipeline reliably guided the fuzzer to every injected bug. Detection rates, however, ranged from only 20% to 42%, demonstrating that reaching a vulnerable code path does not guarantee successful detection. Echidna must additionally construct the precise transaction sequence that triggers re-entry before state is updated, which it achieves inconsistently.

Configuration intensity had a variant-dependent effect. Single-function detection improved consistently with intensity, while cross-function detection peaked at Medium (42%) and declined under Heavy (36%), suggesting that longer sequences introduce exploration noise that disrupts the specific call ordering required for cross-function exploitation.

Temporal analysis showed that all detections occurred within the first 25 seconds of each campaign, with no additional detections gained thereafter. Extended timeout budgets offer no practical benefit for reentrancy detection under this setup. Future work should explore hybrid approaches combining fuzzing with symbolic execution to improve exploit-sequence construction, particularly for cross-function patterns.

ACKNOWLEDGMENT

The authors gratefully acknowledge Trail of Bits for developing and maintaining Echidna, whose open design and extensible configuration made it well-suited for the controlled experimental evaluation presented in this study.

The bug injection pipeline developed for this research is publicly available at https://github.com/hzqula/hz-injectrancy. The tool automates contract instrumentation and reentrancy variant injection, and is released openly to support reproducibility and to facilitate future evaluations of dynamic analysis tools under controlled ground-truth conditions.

REFERENCES

1. A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, "A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks," Front. Comput. Sci., vol. 3, pp. 1–15, Feb. 2021,doi: https://doi.org/10.3389/fcomp.2021.598780.
2. S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H. N. Lee, "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract," IEEE Access, vol. 10, pp. 6605–6621, 2022, doi: https://doi.org/10.1109/ACCESS.2021.3140091.
3. L. Marchesi, L. Pompianu, and R. Tonelli, "Security checklists for Ethereum smart contract development: patterns and best practices," Blockchain: Res. Appl., p. 100367, 2025, doi: https://doi.org/10.1016/j.bcra.2025.100367.
4. Qasse, I. M. Ali, N. Ahmed, and M. Hamdaqa, "The Myth of Immutability: A Multivocal Review on Smart Contract Upgradeability," arXiv:2504.02719, 2025. [Online]. Available: https://arxiv.org/abs/2504.02719
5. AliceHsu, "2024 DeFi Smart Contract Hack Incident Review," Cymetrics Tech Blog, 2024. [Online]. Available: https://tech-blog.cymetrics.io/en/posts/alice/2024_defi_hack/
6. J. V. Behanan and Shashank, "OWASP Smart Contract Top 10," OWASP, 2025. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/
7. S. Burnet and K. Kinder, "Decentralized Finance (DeFi) Market Statistics 2025: TVL, Token Caps & User Adoption Revealed," CoinLaw, 2025. [Online]. Available: https://coinlaw.io/decentralized-finance-market-statistics/
8. S. Cholakov, "2024 Exploits Recap: Top 10 Most Exploited DeFi Vulnerabilities and How to Prevent Them," THREE SIGMA, 2025. [Online]. Available: https://threesigma.xyz/blog/exploit/2024-defi-exploits-top-vulnerabilities
9. A. Ghaleb and K. Pattabiraman, "How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection," in Proc. 29th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), 2020, pp. 415–427, doi: https://doi.org/10.1145/3395363.3397385.
10. J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in Proc. 2nd Int. Workshop Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019, pp. 8–15, doi: https://doi.org/10.1109/WETSEB.2019.00008.
11. Q. Huang, Y. Ju, Y. Jiang, and Y. Shang, "Analysis and Identification of False Positives in Reentrancy Vulnerabilities Based on Anti-Patterns," SSRN, 2024. [Online]. Available: https://ssrn.com/abstract=5024533
12. L. Benetollo, S. Guesmi, C. Piazza, D. Ressi, S. Rossi, and A. Spanò, "Modeling Reentrancy in Smart Contracts through Noninterference," in Proc. Seventh Distributed Ledger Technology Workshop (DLT 2025), 2025. [Online]. Available: https://ceur-ws.org/Vol-4105/paper11.pdf
13. S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, "Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts," in Proc. IEEE/ACM 46th Int. Conf. Software Engineering (ICSE), 2024, pp. 1573–1584, doi: https://doi.org/10.1145/3597503.3639153.
14. Wang, Z., Chen, J., Zheng, Z., Zheng, P., Zhang, Y., & Zhang, W. (2020). Unity is Strength : Enhancing Precision in Reentrancy Vulnerability Detection of Smart Contract Analysis Tools. IEEE Transactions on Software Engineering, 1–12. https://doi.org/10.48550/arXiv.2402.09094.

15. F. R. Vidal, N. Ivaki, and N. Laranjeiro, "Detection techniques for smart contracts: A systematic literature review," J. Syst. Softw., vol. 217, p. 112160, Jul. 2024, doi: https://doi.org/10.1016/j.jss.2024.112160.
16. Y. Xue and Y. Lin, "Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts," in Proc. IEEE Int. Conf. Automated Software Engineering (ASE), 2020, doi: 10.1109/ASE.2020.00029.
17. G. Grieco, W. Song, A. Cygan, and A. Groce, "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts," in Proc. 29th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), 2020, pp. 20–23, doi: https://doi.org/10.1145/3395363.3404366.
18. M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, and C. Sun, "Empirical Evaluation of Smart Contract Testing: What Is the Best Choice?" in Proc. 30th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA), 2021, pp. 566–579, Doi: https://doi.org/10.1145/3460319.3464837.
19. N. F. Samreen and M. H. Alalfi, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," in Proc. Int. Workshop Blockchain Oriented Software Engineering (IWBOSE), 2020, Doi:https://doi.org/10.1109/IWBOSE50093.2020.9050260.
20. N. Deshpande and D. Rana, "Demystifying Reentrancy Attacks on Smart Contracts: Understanding Types and Mitigations," NFSU – J. Cyber Security and Digital Forensics, pp. 66–77, 2024. [Online]. Available: https://jcsdf.nfsu.ac.in/articles?id=54
21. The Solidity Authors, "Security Considerations," Solidity Documentation, 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.24/security-considerations.html
22. Q. Song, H. Huang, X. Jia, Y. Xie, and J. Cao, "Silence False Alarms: Identifying Anti-Reentrancy Patterns on Ethereum to Refine Smart Contract Reentrancy Detection," in Proc. Network and Distributed System Security (NDSS) Symp., Feb. 2025.
23. K. Li et al., "Static Application Security Testing (SAST) Tools for Smart Contracts: How Far Are We?" Proc. ACM Softw. Eng., vol. 1, pp. 1–24, Jul. 2024, doi: https://doi.org/10.1145/3660772.
24. P. E. Black, A. Delaitre, D. Cupif, G. Haben, A.-K. Loembe, V. Okun, and Y. Prono, "SATE VI Report: Bug Injection and Collection," U.S. Department of Commerce, NIST SP 500-341, 2023, doi: https://doi.org/10.6028/NIST.SP.500-341.
25. F. Salzano et al., "An Empirical Analysis of Vulnerability Detection Tools for Solidity Smart Contracts Using Line Level Manually Annotated Vulnerabilities," arXiv:2505.15756, 2025, doi: https://doi.org/10.48550/arXiv.2505.15756.
26. H. Wang, Y. Liu, Y. Li, S. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-Supported Dynamic Exploit Generation for Smart Contracts," IEEE Trans. Dependable Secure Comput., 2022, doi: https://doi.org/10.1109/TDSC.2020.3037332.
27. Imran Pollob, "smart-contract-vulnerability-dataset". https://github.com/imranpollob/smart-contract-vulnerability-dataset/