DeadDrop: Responsible Disclosure of Smart Contract Bugs

Mariarosaria Barbaraci*
University of Bern
Bern, Switzerland
mariarosaria.barbaraci@unibe.ch

Stephanie Ma*
Cornell University
Ithaca, NY, USA
ym363@cornell.edu

Harjasleen Malvai*
UIUC, IC3
Champaign, IL, USA
hmalvai2@illinois.edu

Marwa Mouallem*
Technion
Haifa, Israel
marwamouallem@gmail.com

Silei Ren*
Cornell University, IC3
Ithaca, NY, USA
sr2262@cornell.edu

Yoshi Sato*
Waseda University, Yale
Tokyo, Japan
yoshi.sato@fuji.waseda.jp

Sen Yang*
Yale University, IC3
New Haven, CT, USA
sen.yang@yale.edu

Fan Zhang*
Yale University, IC3
New Haven, CT, USA
f.zhang@yale.edu

Abstract

Modern software security increasingly relies on bug bounty programs, which incentivize independent researchers to discover and responsibly disclose vulnerabilities. While effective in traditional software ecosystems, these programs face unique challenges in the context of smart contracts. Smart contracts run on blockchains, manage financial assets, and are typically authored by pseudonymous developers. As a result, when vulnerabilities are discovered, security researchers often lack a secure and reliable channel for disclosure. Existing workarounds, such as encrypting messages to contract authors' public keys, introduce inefficiencies and privacy risks, since ciphertexts themselves may reveal the presence of bugs.

We propose DeadDrop, the first system to support oblivious bug reporting for smart contracts. DeadDrop combines oblivious message retrieval (OMR) with a trusted execution environment (TEE) to enable researchers to privately and efficiently deliver vulnerability reports without revealing their intended recipient or overwhelming authors with spam. Our design specifies security requirements for such a system, presents practical techniques for handling long messages, and introduces a bug specification language to formalize submissions. We implement a prototype and evaluate its performance, demonstrating that oblivious bug reporting is both feasible and efficient, achieving an amortized processing time of approximately 3 ms per submission. Finally, we discuss incentive mechanisms to encourage participation, highlighting open challenges for decentralized bug bounty ecosystems.

Keywords

Vulnerability Disclosure, Smart Contract, Oblivious Message Retrieval, Trusted Execution Environment

1 Introduction

As modern software systems become increasingly complex, *proactive* security analysis, testing, and auditing cannot uncover all vulnerabilities. As a result, software vendors have turned to *bug bounty programs* as a post-active security measure. Bug bounty programs incentivize independent security researchers to discover and responsibly disclose vulnerabilities by offering monetary rewards or recognition. By harnessing diverse perspectives and adversarial creativity, bug bounty programs have proven effective and been widely employed in practice [16, 19, 28, 29].

However, enabling responsible disclosure for *smart contracts bugs* faces unique challenges. For background, smart contracts are programs that run on blockchains, underpinning critical services such as Decentralized Finance and stablecoins. Perhaps even more so than traditional software, developing bug-free smart contracts is tricky, even with the help of modern analysis tools, partially because of the limitations of existing programing languages, but also the highly adversarial environment in which they run (e.g., anyone can observe a smart contract's code and state and attempt to interact with it), and the significant financial motivation for hackers (e.g., in H1 2024, on-chain security incidents averaged $2.93M in losses per incident; median $231K [9]). Thus, post-active security measures to enable timely reporting and fixing of smart contract vulnerabilities appear particularly important and attractive.

Unlike traditional software, where disclosure usually happens through company security portals, bug bounty platforms, or published contact information, smart contract authors are typically *pseudonymous*. In most cases, the only visible point of contact is the developer's blockchain address. When security researchers discover vulnerabilities in deployed smart contracts, responsibly reporting them is often difficult. Recent academic papers [34, 47] that have discovered critical bugs documented the challenge of reaching the developers of smart contracts, such as *"Unfortunately, all the vulnerable contracts we identified appear to be anonymously deployed and there is no apparent means to identify an entity or person behind these contracts."* [34].

Strong trust in existing bug-bounty platforms. Centralized bug-bounty platforms (e.g., Immunefi and HackenProof) exist for smart contracts but require unpalatable trust, as they gain visibility into readily monetizable zero-day vulnerabilities that anyone can anonymously exploit.

*Authors are listed in alphabetical order.

The lack of privacy is not only an engineering limitation but also a direct consequence of these platforms providing *triage-as-a-service*: they filter, validate, and manage incoming vulnerability reports amid substantial noise. Prior work finds that fewer than 25% of submitted reports are valid across major platforms [49], and HackerOne reports an average validity of 19% [18]. Naively encrypting bug reports would eliminate this trust assumption but also disable filtering, overwhelming developers with spam and invalid bugs. An adversary could spam the service to delay the developer's access to valid reports to gain time to execute the attack.

An orthogonal challenge is to ensure privacy during bug delivery. In order for the platform to deliver an encrypted bug report to the developer, the platform needs to know the recipient of a bug report. It can thus infer which smart contracts are potentially buggy, a critical piece of security information that should not be leaked to any entity besides the developer herself. Hiding the recipient from the platform comes at a cost, as the developers must scan every incoming ciphertext to find the ones they can decrypt, which does not scale and exacerbates the spam attacks.

To summarize, existing bug bounty platforms require strong trust assumptions that are particularly unpalatable for smart contracts.[1] This inherent tension between the privacy of bugs and the performance of spam prevention and retrieval motivates DeadDrop, the first end-to-end private bug-disclosure system for smart contracts.

Resolving privacy and retrieval performance. To enhance privacy, we leverage the Oblivious Message Retrieval (OMR) primitive, which allows a *sender* to post a bug on a public bulletin board while hiding the intended *receiver* of the submission. This is achieved by creating and attaching *clues* to the encrypted bug submission. Meanwhile, the receiver can use such clues and a separate key to find relevant bugs efficiently by offloading computation to a semi-trusted third party. The off-the-shelf implementation of OMR does not support long messages. To solve this issue, we modify the protocol by running only its first phase, which we later define as the *detection mode*. This modification does not affect the security guarantees of the OMR protocol. Additionally, as OMR transmits messages in the clear, we encrypt them before applying OMR, ensuring *confidentiality*.

Spam prevention. DeadDrop mitigates spam through an invariant-based bug validation module executed inside a TEE. We rely on TEEs for their strong confidentiality guarantees and remote attestation capabilities. A submitter who discovers a bug executes the exploit against a provided invariant within the TEE to demonstrate that the invariant can be violated. Using a Merkle tree and a digital signature, the TEE produces an attestation that (i) proves the validity of the submitted bug, (ii) ensures that the bug breaks one of the registered smart contracts, without disclosing which one, and (iii) binds such attestation to the submitted bug.

Automated validation requires contract owners to formally specify correctness conditions, and bug reports must provide runnable implementations that demonstrate violations of these conditions. A specification language must strike a balance between simplicity and expressiveness to meet the diverse needs of different smart contract developers. Looking ahead (§5), DeadDrop assumes a modular and configurable interface for bug validation, while showcasing practical instances, including Solidity-based testing and predefined invariant testing.

Incentives. We design an incentive mechanism that goes hand in hand with the system model and solves some of the security issues through a game-theoretic approach. We model the incentive mechanism as a Tullock contest [44] with incomplete information, where parties hold only *beliefs* on the efforts others put into the search for bugs. This model is valid because of the inherent confidentiality and obliviousness provided by the system model, i.e., every participant (submitter) has no way of knowing what efforts other participants are putting into the contest. Therefore, analyzing the equilibrium, we find that our design incentivizes participation while simultaneously discouraging coalitions or Sybil attacks, i.e., strategies where an agent creates multiple identities or coordinates with others under a winner-takes-it-all strategy. The uncertainty about the presence of other participants additionally makes Tullok's assumptions, like simultaneity of participation and probabilistic choice of the winner, practically feasible. Until the contest is closed and the winner revealed, the order of submission is unknown.

Implementation and evaluation. We implement DeadDrop through two key components: a modified OMR protocol and a TEE-based bug validation module. To integrate the native C++ OMR implementation with our Rust prototype, we refactor its codebase to expose the detection mode via an API while preserving the original cryptographic logic. In our evaluation, processing 32,768 submissions on a single core takes 108 seconds, while scaling to 524,288 submissions across 16 cores yields a similar total runtime (110 seconds), demonstrating near-linear throughput scaling with core count. We build our TEE implementation on top of Phala Network's Dstack framework [52], which offers the primitives required to support practical bug validation. To illustrate the modularity of our specification interface and the practicality of automated validation, we implement representative bug conditions, bug reports, and validation scripts using Foundry—specifically Anvil, a high-performance local Ethereum node, and Forge, Foundry's testing and scripting framework [41, 42].

1.1 Our Contributions

Our contributions are as follows:

(1) We are the first to introduce the oblivious bug bounty system and formalize its security requirements under the universal composability (UC) model, specifying both security and privacy guarantees under a semi-honest server and potentially malicious bug submitters.
(2) We design DeadDrop, a practical realization of the oblivious bug bounty model that combines a cryptographic primitive (OMR) and a TEE to enable secure and privacy-preserving bug submission and retrieval. Specifically, we extend the OMR protocol to efficiently support our setting without weakening its security guarantees, and develop a specification interface that enables automatic bug validation within a TEE.
(3) We implement DeadDrop by integrating a TEE-based validator with our modified OMR protocol. Our evaluation shows

[1] They are also unpalatable for traditional programs, which may explain why large companies run their own bug bounty programs. Our techniques apply to that setting as well, although we focus on smart contracts.

that the protocol is embarrassingly parallel across batches of clues, demonstrating the feasibility of practical oblivious bug reporting.

(4) We design an incentive mechanism that encourages participation and shortens the expected time to discover bugs. We demonstrate through game-theoretic analysis that it is robust to Sybil and collusive manipulation. The incomplete information assumption based on *beliefs* aligns perfectly with the DeadDrop's oblivious reporting through a public bulletin board.

2 Background

2.1 Oblivious Message Retrieval (OMR)

The Oblivious Message Retrieval (OMR) [23–25] primitive allows users of a messaging system to outsource the task of fetching relevant messages (those addressed to the user) without leaking their indices. Specifically, we use the same model in Liu et al. [23], which works as follows: *senders* post messages intended for *receivers* on a public bulletin board BB. Such messages are composed of two parts: payload and clue, where clues are produced using a receiver-specific *clue key*, $\mathsf{pk}_{\mathsf{clue}}$. Receivers can outsource the fetching up to $\bar{k}$ pertinent messages to a semi-honest third party, a *detector*, by providing a detection key, $\mathsf{pk}_{\mathsf{detect}}$. Finally, a detector can retrieve the pertinent messages in the form of a compressed digest that the receiver can later decrypt (decode) with its private key, $\mathsf{sk}_{\mathsf{OMR}}$, retrieving the set of pertinent messages PL. Assuming public parameters pp are always as an input, an OMR scheme encompasses the following algorithms:

- $(\mathsf{sk}_{\mathsf{OMR}}, \mathsf{pk} = (\mathsf{pk}_{\mathsf{clue}}, \mathsf{pk}_{\mathsf{detect}})) := \mathsf{KeyGen}()$.
- $\mathsf{clue} := \mathsf{GenClue}(\mathsf{pk}_{\mathsf{clue}}, \mathsf{payload})$.
- $\mathsf{digest} := \mathsf{Retrieve}(\mathsf{BB}, \mathsf{pk}_{\mathsf{detect}}, \bar{k})$.
- $\mathsf{PL} := \mathsf{Decode}(\mathsf{digest}, \mathsf{sk}_{\mathsf{OMR}})$.

The threat model assumes potentially malicious senders and receivers, as well as a network adversary that can hinder the protocol by delaying, injecting, or dropping messages. We underline the *computational privacy* property, which ensures that a clue does not leak any information about the specific $\mathsf{pk}_{\mathsf{clue}}$ used to generate it. As a result, the information publicly available on the BB and the operations performed by a detector cannot lead to the intended receivers, i.e., $\mathsf{pk}_{\mathsf{clue}}$ and $\mathsf{pk}_{\mathsf{detect}}$ are unlinkable. Overall, OMR provides senders with an easy way to post messages for receivers, and receivers to fetch pertinent messages obliviously, by outsourcing the expensive computation to the detector.

2.2 Trusted Execution Environment

Trusted Execution Environments (TEEs) are hardware-isolated regions within a processor that ensure the confidentiality and integrity of code and data against privileged software. Modern TEEs such as AMD SEV-SNP and Intel TDX extend this protection to virtual machines (VMs) [2, 11], creating confidential VM (CVMs) that are isolated execution environments that protect entire guest operating systems from an untrusted host. These CVMs provide hardware-enforced memory encryption, cryptographic measurement of VM state, and remote attestation that allows external verifiers to confirm the authenticity of the execution context.

For bug bounty platforms, these guarantees are critical. Confidential execution allows vulnerabilities to be tested and validated without exposing exploit details to the platform or host, while attestation ensures that validation occurs in a trusted environment. However, this VM-level attestation remains coarse-grained: it attests to the entire VM image rather than the specific validation logic or containerized workload. Recent confidential container frameworks address this limitation by supporting fine-grained container-level attestation, allowing the platform to verify the exact validation module and cryptographically bind its output to an attestation report. This capability forms the foundation for DeadDrop's TEE-based bug validation module.

2.3 Merkle Tree

Merkle trees [27] are hash-based authenticated data structures that commit to a set of n leaves with a short commitment, com (the root hash). Given a Merkle tree and one of the items committed in the set, a party can generate a proof π of length $\log n$, using the $\mathsf{Prove}_{\mathcal{M}}$ function. Given a commitment com and a proof π, a party can verify that some item e is in the original committed set by running $\mathsf{Verify}_{\mathcal{M}}(\mathsf{com}, e, \pi)$. In DeadDrop, we use Merkle trees to commit to the current set of smart contracts registered to the system, including the conditions for a valid bug, as defined by the contract's developer. The TEE-based bug validation module hides the actual contract in which a bug has been found. In this case, the server must prevent spam by ascertaining that an attestation verifies that the bug with respect to a contract (and invariants) that are actually registered with the system. To this end, the attestation includes a check $\mathsf{Verify}_{\mathcal{M}}$ and provides the commitment, com, used as output viewable by the server. The server compares this com against its own commitment and only accepts a submission if the attestation passes and these commitments match.

3 Threat Model

Here, we discuss our threat model for oblivious bug submission. We clarify terminology (§3.1), introduce system entities (§3.2), and finally propose an abstract model of the system through the definition of an ideal functionality (§3.3).

3.1 Terminology

We use the term *bug* in an abstract sense. In practice, we model it as a tuple consisting of a contract state and a sequence of transactions or commands made to the contract that drive the contract into an undesirable state.

A *test suite* for a smart contract is defined as a collection of tests, each checking an invariant, i.e., a predicate on the contract's behavior. A bug is deemed valid if the contract begins in a state that satisfies an invariant, yet after executing a sequence of transactions (as provided by a bug-bounty hunter), the invariant no longer holds. In this paper, we sometimes call the test suite a *condition*, denoted cond, where failing some test is equivalent to the condition cond outputting 0 on a bug.

3.2 System Entities

The system encompasses the following parties:

- **Receiver**: Smart contract developer who publishes calls for bug reports for their smart contracts.
- **Submitter**: Bug searcher who looks for bugs in a smart contract and ideally submits only *valid* bug reports for the smart contract developers.
- **Bulletin Board (BB)**: Publicly accessible ledger which is readable by any receiver or submitter. The write policy for the bulletin board may vary depending on the implementation. For the purposes of this paper, we assume that the reads from the bulletin board are oblivious, i.e., a network adversary may learn that a read request was executed, but not *what* was read.
- **Server**: A semi-honest bug reporting server that acts as an intermediary between the receiver and the submitter: it verifies the bug attestation, and publishes the relevant information on the publicly readable BB.
- **Detector:** A semi-trusted fetcher to which receivers can outsource the monitoring and retrieval of pertinent messages. The detector cannot learn the recipient of messages, but it is trusted for its availability.

Note, however, that in the security analysis we assume an idealized version of the oblivious message retrieval protocol, ignoring the specific implementation mode. Thus, in this section, we do not consider the detector or the bulletin board used in the OMR protocol, which we describe in §4.

3.3 Defining Security

We consider the setting of a smart contract developer who wishes to learn about potential bugs in their contract without running a full-fledged bug bounty program, similar to those operated by large organizations (e.g., Google or Facebook). Instead, the developer may periodically "come online" to check whether any bugs have been submitted for their contract. To facilitate this process, we assume the presence of a server that coordinates interaction between developers (receivers) and bug submitters, as well as any number of independent submitters who may attempt to contribute bug reports.

Crucially, we do not assume the server is trusted with confidentiality: if the server were to learn which contracts contain bugs, it could leak information that exposes vulnerable contracts to additional scrutiny from malicious parties. Even the mere knowledge that a given contract contains an undiscovered bug can incentivize attacks before the developer has had a chance to patch the issue.

Meanwhile, submitters may try to submit bogus bugs and spam the system, thus, the additional responsibility of the server is to verify that submitted bugs are valid.

Security properties.

- Any bug submission accepted by the semi-honest server is valid.
- A receiver should, with high probability, receive all bug submissions meant for it.

Privacy properties.

- Assuming an honest submitter sub and an honest receiver R, if sub submits a bug intended for R, the only sub and R learn the content of the submitted bug. Further, the only parties that learn the intended recipient of this bug are sub and R.

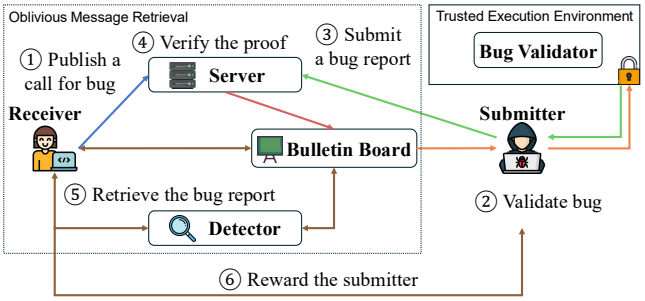

Figure 1: The architecture overview of DeadDrop.

- Assuming an honest receiver, R, the only party that learns whether a set of retrieved bugs was meant for R, is receiver. Further, the only thing learned by any other party is that *some* party retrieved m messages which pertain to it but not which party it was.

Our functionality therefore enforces obliviousness:

- The server only sees encrypted or blinded protocol messages, learning nothing about which contracts are vulnerable or which bugs have been submitted.
- Receivers (developers) only learn about bugs pertaining to their own registered contracts, and learn nothing about the status of other contracts.
- Submitters can verify whether their submissions are accepted (i.e., satisfying the developer's registered conditions), but cannot gain information about unrelated contracts.

4 DeadDrop Protocol

We present the protocol for responsible disclosure of bug reports, DeadDrop, and show an overview in Figure 1.

(1) A smart contract owner willing to receive bug reports for her smart contracts publishes a call for bugs on the DeadDrop server, providing the necessary information for submitters to validate bugs and construct bug reports.
(2) When a security researcher (submitter) finds a bug for an open call, she runs a local TEE-based validator to verify it and produce a proof of the bug's validity.
(3) The submitter submits a bug report to the server, which includes the validity proof, an OMR clue for oblivious retrieval, and the encrypted bug description itself.
(4) DeadDrop server verifies the validity of the bug reports against the conditions set by the developer in the call for bugs. If accepted, the report will be posted for retrieval.
(5) The bug receiver tasks a third-party detector to retrieve pertinent bug reports through the modified OMR protocol.
(6) The receiver rewards a submitter according to a committed strategy, e.g., rewarding the first submitter.

We first introduce the two key modules: a modified OMR implementation for large messages (§4.1) and a TEE-powered bug validation module (§4.2). Then we present the full protocol (§4.3).

4.1 Adapting OMR to DeadDrop

The protocol is built around the Oblivious Message Retrieval (OMR) primitive [23], proposing a modification that enables the practical

use of this primitive in our system. A fundamental constraint of the OMR protocol is that its retrieval mechanism is a Fully Homomorphic Encryption (FHE) circuit that packs all pertinent messages into a single encrypted digest. The capacity of this digest is strictly bounded by the FHE scheme's polynomial modulus degree (i.e., the total number of plaintext slots). This imposes a hard limit on the aggregate size of all retrieved messages. While this is suitable for small, fixed-size payloads, it is incompatible with our use case, as large, variable-length bug reports would easily exceed the entire slot capacity of a single ciphertext, rendering the protocol unusable.

Therefore, we modify the retrieval protocol to distinguish between two modes: *retrieval mode* (default) and *detection mode*. In the detection mode, a detector returns an encrypted digest, which we refer to as report, that contains only the encrypted indices of pertinent messages, effectively halting the default retrieval protocol one step earlier. Thus, we add the following algorithm to OMR:

$$\text{report} := \text{OMR.Detect}(\text{BB}, \text{pk}_{\text{detect}}, \bar{k}).$$

Although this modification requires an additional interaction with the BB to later fetch pertinent messages, it does not affect the security guarantees of OMR when reading from an oblivious BB.

4.2 TEE-based Bug Validator

To prevent spam and denial-of-service attacks caused by malformed bug reports, the server acts as an intermediary between submitters and the bulletin board by validating submissions before they are recorded. Validation leverages a trusted component and exposes two interfaces: (1) a TEE.Proof algorithm, executed inside a TEE, which we call *validator*, run by the submitter, and (2) a TEE.Verify algorithm, executed by the server. The validator ensures that the validation logic executes correctly and that its outputs remain cryptographically tied to the trusted environment.

Given a bug report b, a registered bug condition cond, a secret key pk_{SCenc} associated with the smart contract owner, and the current set of conditions Conds, the TEE executes $\pi_{\text{TEE}} := \text{TEE.Proof}(b, \text{cond}, \text{pk}_{\text{SCenc}}, \text{Conds})$ to determine whether the report satisfies the bug condition and can be posted. If validation succeeds, the output π_{TEE} consists of the application-specific validation artifacts (π_C, π_M, π_E), and the TEE hardware-backed signature σ_{TEE} over these outputs, binding them to the validator code and the particular TEE instance.

The server can verify the attestation by running $\text{TEE.Verify}(\pi_{\text{TEE}}, \text{rt})$ to check that both the attestation and the enclosed proofs are valid. We provide a detailed explanation of each element in the proof.

Correctness proof π_C. The correctness proof ensures that b constitutes a valid bug for condition cond, preventing arbitrary or empty submissions. The validator simulates the exploit against the bug, checks the conditions, and assigns $\pi_C = 1$ if the bug report is valid. For generality, we leave the specification language for bug condition and bug report open. Instead, we define an interface for bug validation and provide examples of implementations in §5.

Membership proof π_M. In our design, we assume that bug conditions Conds are continuously posted to the bulletin board BB. The membership proof π_M claims that a specific $\text{cond} \in \text{Conds}$ holds without revealing cond itself, preventing bug submissions for unregistered contracts. To achieve this, conditions Conds are committed as a Merkle tree, whose order in the tree follows the order they are posted on the BB. The submitter computes the latest Merkle root $\text{rt} = \text{Commit}_{\mathcal{M}}(\text{Conds})$ and a Merkle authentication path $\pi_m = \text{Prove}_{\mathcal{M}}(\text{Conds}, \text{cond})$ to the broken condition. Then it verifies the proof by checking $\text{Verify}_{\mathcal{M}}(\text{cond}, \pi_m, \text{rt}) == 1$, which holds only when cond is a registered contract (by construction of Merkle Tree). Once verified, the validator sets $\pi_M = \text{rt}$. The server can then verify that π_M matches its own rt to ensure the one used for validation is up to date.

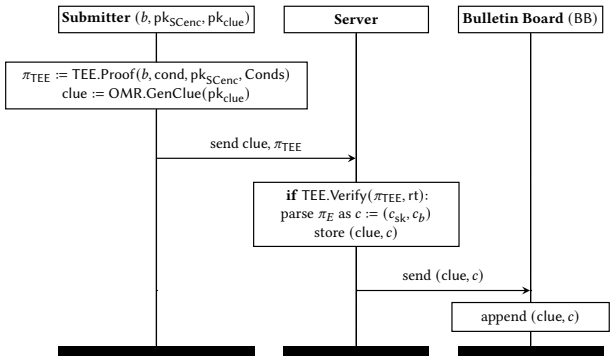

Figure 2: Protocol for submitting bug reports to DeadDrop, using OMR detection mode.

Encryption proof π_E. As the submitter needs to encrypt the bug report before submitting it, she delegates such a task to the TEE validator. In particular, the validator leverages a hybrid-encryption scheme to encrypt the bug report b more efficiently. Thus, he generates a fresh AES key sk and encrypts such key with the receiver's encryption public key pk_{SCenc}, producing c_{sk}. Then, it encrypts b with sk, generating the encrypted bug report c_b. Finally, it assigns $\pi_E := (c_{\text{sk}}, c_b)$. The server will accept such a ciphertext after verifying the attestation.

Integrity σ_{TEE}. Finally, the remote attestation protocol guarantees the integrity of both the validator container binary and its outputs (π_C, π_M, π_E). DeadDrop publishes both the validator code and the compiled binary so that the binary is publicly verifiable. The server verifies the hardware quote, checks that the TEE measurement matches the expected validator code, and validates σ_{TEE} over the outputs before proceeding.

4.3 Putting it All Together

The complete protocol for bug reporting, denoted Π_{DeadDrop}, encompasses two main phases: the submission phase, shown in Figure 2, and the retrieval phase, shown in Figure 3. First, we describe the necessary steps for setup, and then detail each phase, describing the interactions and computations of each entity.

Setup. We assume a setup phase in which the receiver, i.e., smart contract owner, publishes a call for bugs on the BB, providing the necessary *conditions*, cond, (tests and invariants) to check the validity of potential bugs. Additionally, he registers an encryption

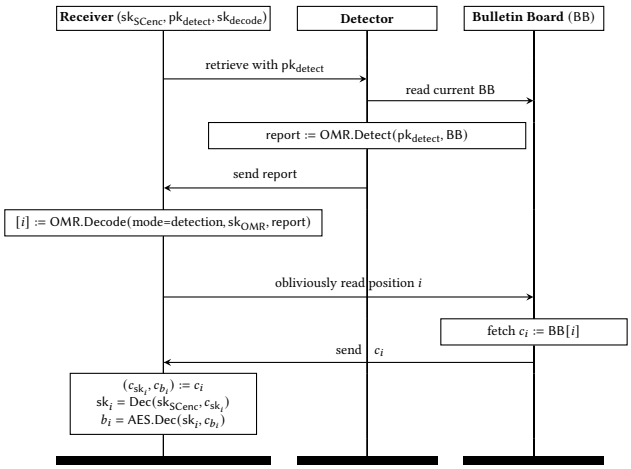

Figure 3: Protocol for retrieving bug reports, using OMR detection mode.

key, pk_{SCenc}, to receive encrypted bug reports, and a clue key, pk_{clue}, to receive such reports obliviously, as an OMR receiver.

Submission. Upon finding a valid bug b for a specific smart contract, a submitter initiates the protocol with the server for submission. The submitter needs to transmit b to the receiver both in an *oblivious* and *private* way. The OMR primitive provides a solution for the first, but it does not allow for big message sizes in the retrieval mode, and confidentiality for the messages. Thus, the submitter will need to encrypt the bug report, b, and uses the resulting ciphertext as payload for OMR used in detection mode. The submitter delegates such tasks to the TEE validator. The TEE-based bug validator encrypts b, and attests it against a specified condition cond, e.g., at least one of the tests in the test suite fails. Therefore, the resulting attestation proof, π_{TEE}, validates both the legitimacy of the bug report and the OMR payload (i.e., the chiphertext). The submitter, following the OMR protocol, computes the relative clue using the receiver's pk_{clue}. Finally, the submitter sends $(\text{clue}, \pi_{\mathsf{TEE}})$ to the DeadDrop server. The server checks the attestation proof, and if valid, after checking the π_C and the π_M, he extracts the ciphertext c from π_{TEE} and sends the OMR tuple (clue, c) to the BB.

Retreival. Whenever the receiver is interested in checking if there are bug reports available on the server, it needs to start the retrieval protocol (fig. 3). The receiver can outsource most of the work to a detector. Upon providing a detection key pk_{detect}, the detector requests the current state of BB and uses OMR in detection mode to filter pertinent entries in the form of a report. Consequently, the receiver can decode the report as follows:

$$[i] := \text{Decode}(\text{mode=detection}, sk_{OMR}, \text{report}).$$

The result is a set of indices of pertinent messages on the BB that the receiver can later fetch. For each message at position i, the receiver parses c_i as (c_{sk_i}, c_{b_i}), decrypts c_{sk_i} to get the key sk_i, and finally decrypts the bug report b_i.

For this paper, we assume that reading from the bulletin board is oblivious, i.e., does not leak access patterns. An adversary therefore learns that k messages were read by someone but not *which* k messages. In practice, access pattern leakage does exist in many blockchains, which are frequently used as bulletin boards; however, we defer the reader to other works (e.g., [6, 26, 40]) for further discussion and potential patches.

Reward the submitter. After the receiver retrieves the report using the modified OMR protocol, they follow the committed strategy to reward the submitter. In this paper, we reward the first submitter to incentivize participation while also discouraging coalitions and Sybil attacks. We provide the incentive analysis in §7.

4.4 Security Analysis

We formalize the security and privacy properties we desire are captured through an ideal functionality, $\mathcal{F}_{ob}$. §3.3 discusses the particular security properties guaranteed by $\mathcal{F}_{ob}$ in the ideal world.

Alternative protocol description. We denote our protocol presented here as Π_{DeadDrop}. In appendix A.2, we present an alternate, modular description of Π_{DeadDrop}, using idealised subcomponents for the oblivious message retrieval ($\mathcal{F}_{\mathsf{OMR}}$), trusted execution environment checking that some bug report is encrypted correctly and breaks an invariant ($\mathcal{F}_{\mathsf{TEE}}$) and an ideal bulletin board ($\mathcal{F}_{\mathsf{BB}}$), which allows a party to post to a bulletin board and also allows parties to read all posts.

Modular component instantiation. Since the components introduced thus far (OMR and TEEs) have been modelled in previous work, in the interest of space, we omit security arguments for them. Note that there are two places where the bulletin board shows up in the description presented in §4—as part of the OMR protocol and to store the set of registered contracts and the required information from their developers. Since we assume an abstracted version of the OMR, we only consider the bulletin board for registering contracts to the bug-bounty system. The following is an example of how the bulletin board could be instantiated. When the system is initialized, the server sets up a smart contract that stores the relevant information (public keys, clue keys, proof of being the developer) for the developer of any contract that registers for the system.

With these abstractions in mind, we present the main theorem:

Theorem 1. *The protocol* Π_{DeadDrop} *emulates* $\mathcal{F}_{ob}$ *in the* $(\mathcal{F}_{\mathsf{OMR}}, \mathcal{F}_{\mathsf{TEE}}, \mathcal{F}_{\mathsf{BB}})$*-hybrid model, assuming honest receivers, a semi-honest server and potentially malicious submitters.*

We provide a sketch of the proof for theorem 1 in Appendix B. We consider honest receivers (smart contract developers), a semi-honest (also called honest-but-curious) server and potentially malicious submitters. We argue that this threat model is realistic for the following reason. A malicious server could simply drop all submissions and the system would cease to function altogether.

5 Bug Validation Specification Language

In this section, we explore the design space of our bug specification and reporting language, which §4.2 relies on, in detail. Designing such a validation language requires balancing simplicity and expressiveness. At one end of the spectrum, bug receivers who value ease of use may prefer a set of simple, predefined conditions. At

Table 1: Summary of variables and data structures in the protocol.

Entity	Name	Description
Receiver	$\mathsf{pk}_{\mathsf{SCenc}}$	Encryption key published as information of a smart contract developer.
	$\mathsf{sk}_{\mathsf{SCenc}}$	Secret key associated to $\mathsf{pk}_{\mathsf{SCenc}}$.
	$\mathsf{pk}_{\mathsf{clue}}$	OMR clue key published as information of a smart contract developer.
	$\mathsf{pk}_{\mathsf{detect}}$	OMR detection key, not linked to the developer identity, therefore to $\mathsf{pk}_{\mathsf{clue}}$.
	$\mathsf{sk}_{\mathsf{OMR}}$	OMR secret key.
	Conds	Set of conditions (or invariants) to check for the developer's smart contracts.
Submitter	TEE Validator	Trusted component delegated to validate the bug b
	b	The bug found.
Server	BB	Bulletin Board with oblivious access. Everybody can read, but only the server has write access.
	rt	Merkle root computed from the set of conditions Conds.
TEE Validator	π_C	Correctness proof that validates the bug against a certain cond $\in$ Conds.
	π_M	Membership proof that ensures the validity of cond as one of the conditions published by the receiver.
	π_E	Encrypted bug report.
	σ_{TEE}	TEE signature on attesting the integrity of the output.

the other end, receivers who wish to capture more nuanced security properties—such as information-flow constraints—require a language capable of expressing formal logical invariants and trace properties [10].

Instead of making a concrete choice of the specification language, we define an extensible and modular interface in §5.1 and describe two implementations of the specification language, accompanied by examples in §5.2 and §5.3.

5.1 Bug Validation Interface

The interface includes the specifications of validation conditions (cond), bug report (b), and a general bug validation logic, which we now explain.

Validation condition (cond). Every bug condition $\mathsf{cond} = (\mathsf{I}, \mathsf{addr}, \mathsf{env}_{\mathsf{bound}})$ consists of an invariant specification I, the address addr of the receiving contract, and a set of environments $\mathsf{env}_{\mathsf{bound}}$ necessary for reproducing the bug (e.g., least acceptable block height and restrictions on block timestamp). The invariant specification expresses as much as common exploit categories as possible; including integer overflow/underflow, access control violations, reentrancy, assert/require failures, denial of service, transaction order dependence, and business logic errors.

Bug report (b). A bug report $b = (\mathsf{txSeq}, \mathsf{env})$ consists of two parts: (1) txSeq, an ordered sequence of transactions that can trigger the bug, (2) env, the execution environment required for simulating the transactions, which includes block height, chain identifier, fork tag, and block header fields, etc.

Bug validation logic. Given a report submission $b = (\mathsf{txSeq}, \mathsf{env})$ from a submitter and a validation condition $\mathsf{cond} = (\mathsf{I}, \mathsf{addr}, \mathsf{env}_{\mathsf{bound}})$ from the receiver, the TEE validator first checks that the initial state of the exploit is met ($\mathsf{env} \in \mathsf{env}_{\mathsf{bound}}$). Then it verifies if addr is relevant to the attack by checking whether txSeq contains (1) a direct call to addr, (2) or a call executed under addr's delegation, or (3) or a read/write to addr's state. The validator then initializes an execution environment at block height B specified byenv, forks the on-chain code and state necessary to reply txSeq, and monitors for the invariants. After the checks are performed, the validator replays the transactions. π_C is set to 1 if and only if at least one invariant in I is violated during the replay.

5.2 Generic Invariant Validation

For bug receivers who value simplicity over expressiveness, DeadDrop offers a predefined set of common validation conditions cond that can be directly activated, relieving them from manually defining conditions. The submitter then provides a list of raw transactions as the bug report b to reproduce potential exploits. The system is implemented in Rust and relies on Anvil [41] as the backend execution environment, where the validator executes transactions and checks for invariant violations. This implementation supports generic checks, including asset security (asset conservation), access control issues (immutability of privileged variables such as `owner` or `admin`), and temporal invariants (preventing state updates after a specified time).

Example. As an illustrative example written in pseudocode, consider an asset-security invariant, which ensures that a contract's ERC-20 token balances remain unchanged at a given block height. During validation, the validator initializes an execution environment at the given block height and queries each token contract's `balanceOf(addr)` before executing txSeq (using `WETH` as the illustrative token). It then re-queries these balances after execution. If any token balance decreases, the invariant is violated, and the validator outputs $\pi_C = 1$, indicating that the submitted transactions successfully trigger an asset-draining vulnerability.

```
fn asset_security_invariant(addr: Address, txSeq: Vec<Transaction>)
    {
    // initialize env
    // check balances per predefined invariants
    let pre = query_balance(addr, WETH);
    // execute txSeq in the forked state
    execute(txSeq);
    let post = query_balance(addr, WETH);
    // invariant: balance must remain unchanged
    assert_eq!(pre, post);
}
```

Listing 1: Asset Security Invariant

5.3 Test Suite for Validation

A more expressive instantiation of the bug validation interface is using Foundry's Forge [42] testing framework. Foundry Forge is a modular testing framework for Ethereum that supports local deployment for developing smart contracts. In Forge-based testing, the validation condition cond is a set of Forge test files with assertions specified by bug receivers. A bug report b is an implementation of the bug again as a Forge test file that replays the transaction sequence. The bug validator imports the bug report with the designated validation condition and executes the Forge tests in the TEE environment. The validation bit π_C is set to 1 if any of the Forge tests fail.

Example. We illustrate the expressiveness of Forge tests as a specification language using the classic "King of the Hill" (KoTH) example. KoTH is a well-known design pattern for reasoning about competition and liveness in smart contracts. The pattern maintains a single "king", who can be replaced by anyone paying a higher bid, thereby modeling a simple bidding logic frequently used in early Ethereum games and contracts.

```
1  contract KoTH {
2      address public king;
3      uint public prize;
4
5      function claimThrone() external payable {
6          require(msg.value > prize);
7          payable(king).transfer(prize);
8          king = msg.sender;
9          prize = msg.value;
10     }
11 }
```

Listing 2: KoTH Vulnerable Contract

Listing 2 shows the vulnerable `claimThrone()` function (lines 5–10). It first refunds the previous king via `transfer` (line 7) and only then updates `king` and `prize`. Because the refund occurs before the state update, the throning process is blocked if the refund fails. This creates a liveness issue: once the current king cannot accept payments, no one else can claim the throne.

Using Forge, the validator runs the `validate` function from lines 19-23 in listing 4 that combines the pre-condition and post-condition of the receiver and the attack of the receiver submitter.

```
1  contract BadKing {
2      receive() external payable {
3          revert("no refund");
4      }
5
6      function play(address koth) external payable {
7          koth.claimThrone{value: msg.value}();
8      }
9  }
10
11 contract BugReport is Test {
12     function attack(address addr) internal {
13         BadKing attacker = new BadKing();
14         vm.prank(address(attacker)); // adopt role
15         uint bid = KoTH(addr).prize() + 1 wei;
16         vm.deal(address(attacker), bid);
17         BadKing.play{value: bid}(addr);
18     }
19 }
```

Listing 3: KoTH Bug Report

Listing 3 shows a bug report provided by the submitter. The `BadKing` contract rejects all incoming payments (lines 2–4), ensuring that any attempt to refund will revert. The `BugReport` contract then makes the attacker claim the throne with a slightly higher bid (lines 12-18). After these steps, any new bidder attempting to dethrone the king will fail.

Liveness conditions cannot be described by simple assertions over the contract state. In Listing 4, the validator adopts the role of a new bidder and sends a qualifying bid to `claimThrone()` (line 7-13). The invariant tests two expected outcomes: the call succeeds (line 15) and the bidder is successfully throned (line 16). If either assertion fails, the liveness condition is violated and π_C is set to 1, indicating that the submitted bug report exposes a vulnerability of denial-of-service.

```
1  contract KoTHConditions is Test, BugReport {
2      KoTH koth;
3
4      function pre_inv() internal { }
5
6      function post_inv() internal {
7          address bidder = makeAddr("bidder");
8          vm.prank(bidder);
9          uint bid = koth.prize() + 1 wei;
10         vm.deal(bidder, bid);
11         (bool ok, ) = koth.call{value: bid}(
12           abi.encodeWithSignature("claimThrone()")
13         );
14
15         assertTrue(ok);
16         assertEq(koth.king(), bidder);
17     }
18
19     function validate() public {
20         pre_inv();
21         attack(address(koth));
22         post_inv();
23     }
24 }
```

Listing 4: KoTH Liveness Condition

6 Evaluation

In this section, we report the performance evaluation of a proof-of-concept implementation for the protocol presented in §4. Our implementation includes a bug reporting server, which stores encrypted bug reports upon attestation verification, and from which such bugs can be retrieved. We also implemented the OMR primitive, which is used by the receiver, the submitter, and the detector.

6.1 Implementation Details

We implement DeadDrop with three main components: a web frontend (Typescript/HTML) for submitters and receivers, a Rust-based validator that verifies a bug against an invariant inside a TEE, and a cryptographic backend that operates the OMR logic.

Oblivious message retrieval. Our cryptographic backbone builds on the recent DoS-resistant OMR construction from Snake-eye Resistant PKE [23], which extends and strengthens the PerfOMR protocol [25]. We refactored the authors' monolithic test harness into callable API functions for modular integration. The main structural change was isolating the detection phase of the original retrieval mode as a standalone detection mode. We also

parallelized the precomputation phase to significantly reduce the initialization latency and improved the practicality of the OMR prototype for real-world use. In the evaluation below, we consider this preparation step *offline* because it only needs to be performed once for each client public key, regardless of how many encrypted clues are later processed. Separately, we also consolidated scattered key-generation steps into a single initialization function that materializes the three main keys required by the system.

We integrate OMR, implemented in C++, into our Rust prototype with an HTTP-based service layer.

Signatures and encryption schemes. All of our encryption is implemented in Rust, as it must be attested by the TEE-bug-validation module. In particular, for the symmetric encryption, we use the following crates: aes [38] (v. 0.7.5), block-modes [35] (v. 0.8.1) and block-padding [36] (v 0.2). For public key encryption, we used the age [22] (v 0.10) Rust crate.

Merkle Tree. We use a simple static Merkle tree implemented in Rust with the `sha2` crate [37] (`v0.10`).

TEE for bug validator. We implement the validation module as a Docker image and use the Dstack SDK [13] to deploy it within a confidential virtual machine (CVM). The CVM runs on an Intel TDX server hosted by Phala Network [32]. Ideally, the server-side verifier would programmatically validate attestations in real-time, as illustrated in Figure 2. However, Phala Network does not currently support automated remote attestation verification for third parties. Instead, verification relies on the Phala Attestation Explorer [33], a web-based tool requiring manual interaction. Consequently, our implementation adopts a hybrid approach: the client manually facilitates the process and submits the resulting metadata, which the server uses to retrieve and verify the attestation record from the Explorer. Integrating a fully automated TEE verification SDK remains future work.

6.2 Evaluation

Setup. All experiments were conducted on Google Cloud instances (`n4-standard-16`, 16 vCPUs, 64 GB memory) provisioned in the `us-central1` region. The VMs ran Debian GNU/Linux 12 ("bookworm") with Linux kernel 6.1.0-37-cloud-amd64 on Intel Emerald Rapids CPUs.

We evaluated the prototype primarily in terms of the runtime (latency) of the primary functions in the OMR pipeline, since this reflects the dominant cost affecting system usability. Runtime refers to the end-to-end execution time of each primary function (e.g., clue generation, report generation, and report decoding). Bandwidth is not a limiting factor in our setup because the exchanged ciphertexts are small compared to the computation cost.

We measured the function runtimes using the C++ clock API (`std::chrono::high_resolution_clock`) to record wall-clock time between execution start and completion. Each reported value is the mean of 10 runs, preceded by three warm-up runs that are excluded from analysis. All timings exclude key generation but include all other cryptographic operations.

Results. Table 2 reports the runtime of the four primary OMR functions in DeadDrop, measured on 32,768 clues. GenReport(online) measures the server-side lookup over the encrypted clues, while

Table 2: Runtime for individual functions

Function	Average Time (s)
GenClue()	0.0147
GenReport(online)	108.0
GenReport(offline)	108.0
Decode()	0.0199

Table 3: Bandwidth/size of components

Function	Serialized Size
$\mathsf{sk}_{\mathsf{decode}}$	4.250 Mb
$\mathsf{pk}_{\mathsf{clue}}$	5.450 MB
$\mathsf{pk}_{\mathsf{detect}}$	24.00 MB
clue	7.344 KB
encrypted report	1.000 MB
report	0.250 MB

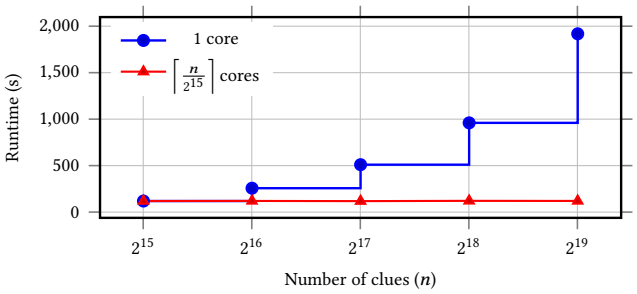

Figure 4: Runtime for generating the report as the number of clues increases. The runtime grows in left-continuous discrete steps because clues are processed in fixed-size blocks of 2^{15} slots, and any n that is not a power of two is truncated to the nearest supported size. The lookup is parallelizable across blocks, keeping runtime roughly constant when both the number of clues and cores double.

GenReport(offline) corresponds to the precomputation step described in §6.1. Compared to clue generation and report decoding, the lookup phase dominates total runtime because each homomorphic operation involves large-degree polynomial arithmetic, modular reductions, and key-switching transformations. These operations make FHE computation several orders of magnitude slower than plaintext processing, and thus the observed runtime reflects intrinsic cryptographic cost rather than data volume.

We process 32,768 clues because this number corresponds to the polynomial modulus degree used by the underlying OMR library, which parameterizes each ciphertext to contain a vector of 32,768 slots (one per clue), and homomorphic operations act on all slots simultaneously. Consequently, 32,768 clues constitute one batch in single-threaded execution, while larger workloads are split across multiple ciphertexts that can be processed in parallel.

The step pattern in Figure 4 stems from the batching constraint imposed by the polynomial modulus degree. When the number of

clues is not an exact multiple of this parameter, the OMR implementation truncates to the nearest supported block size, yielding the left-continuous discrete steps shown in the figure. In the single-core case, runtime scales linearly with the number of clues, whereas increasing the number of cores proportionally keeps runtime nearly constant. With 16 blocks of clues (2^{19} or 524,288) and 16 cores, the runtime is 110 s. We evaluate up to 524,288 clues, the maximum batch size supported by our ciphertext encoding under the chosen homomorphic encryption parameters. As shown in Table 3, the report size remains constant regardless of the number of clues processed. Increasing this limit and enabling further parallelization would require outputting multiple ciphertexts.

Overall, the scaling behavior reveals that OMR's computation model is ciphertext-centric: performance scales with the number of ciphertext batches rather than individual clues. Each batch represents a vector of encrypted clues defined by the polynomial modulus degree, making the workload embarrassingly parallel across ciphertexts. As a result, OMR achieves predictable linear scaling and high parallel efficiency, yet its throughput remains bounded by the latency of per-ciphertext FHE operations.

7 Incentives Design and Analysis

To incentivise searchers to submit their discovered bugs, receivers offer rewards for successful submitters. In this section, we design the incentive mechanism to maximize submitters' participation and minimize the time needed for bug discovery. First, we model the system as a Tullock contest, characterize equilibrium effort under incomplete information, and show robustness to Sybil and collusive manipulation (§7.1). Second, we link searchers' efforts to the expected bug discovery time and examine how parameters such as the number of participants and bounty size affect performance (§7.2). Finally, we discuss how the game informs our protocol design (§7.3).

7.1 Model and Equilibrium Characterization

We model the bug-bounty competition in DeadDrop as a Tullock contest [44]. A set of n submitters $\mathcal{S} = \{1, \dots, n\}$ compete for a single bounty $V > 0$ posted by a receiver. The bug bounty remains open until a submitter submits a valid bug report. Each submitter i chooses a nonnegative search intensity $b_i \in \mathbb{R}_{\geq 0}$ and incurs a linear effort cost at a privately known rate. The bounty V is paid to the first submitter to find and report a valid bug.

Let $b = (b_1, \dots, b_n)$ be the effort profile, which implies a total search intensity $B = \sum_{j=1}^{n} b_j$. When $B > 0$, the contest success function (CSF) is

$$p_i(b) \;=\; \frac{b_i}{B},$$

that can be interpreted as the probability that submitter i is the first to report a valid bug. A submitter may opt not to search by choosing $b_i = 0$, in which case $p_i(b) = 0$ whenever some opponent exerts positive effort. When $B = 0$, the bounty is allocated uniformly at random across $\mathcal{S}$. Since $V > 0$, the all-zero profile is off-equilibrium because any unilateral deviation to $\varepsilon > 0$ yields a strictly positive expected payoff.

Each submitter i has a privately known marginal search cost $c_i \in \Theta \subset (0, \infty)$. Costs are drawn i.i.d. from a common distribution F on Θ with density $f > 0$, and the distribution F is common knowledge. The distribution F captures shared beliefs about the cross-sectional distribution of marginal search costs in the participant pool, while each cost c_i remains private.

Given a profile $b = (b_1, \dots, b_n)$ and a cost c_i, submitter i's expected payoff is

$$u_i(b, c_i) \;=\; V\,\frac{b_i}{B} \;-\; c_i\, b_i \qquad (B > 0),$$

extended naturally under the tie-breaking rule when $B = 0$.

The game progresses as follows: First, nature draws $c = (c_1, \dots, c_n)$ with $c_i \overset{\text{i.i.d.}}{\sim} F$ and privately reveals c_i to submitter i. Submitters then simultaneously choose search intensities $b_i \in \mathbb{R}_{\geq 0}$. Finally, the bounty is allocated according to $p(\cdot)$ and payoffs realized.

Equilibrium. Having defined the players, payoffs, and timing of the game, we next characterize how rational submitters choose optimal effort levels under incomplete information. A pure strategy is a measurable policy $\beta : \Theta \to \mathbb{R}_{\geq 0}$ that maps a privately known cost c_i to a search intensity, $b_i = \beta(c_i)$. By i.i.d. costs and common knowledge of F, we focus on symmetric Bayesian Nash equilibrium (BNE) in which all submitters use the same policy function β. Let $C_1, \dots, C_{n-1} \overset{\text{i.i.d.}}{\sim} F$ and let $S = \sum_{k=1}^{n-1} \beta(C_k)$ denote the random total search intensity of opponents. A policy β is a symmetric BNE if, for every cost $c \in \Theta$, it solves

$$\beta(c) \in \arg\max_{b \geq 0} \; \mathbb{E}\left[\, V\,\frac{b}{b + S} - c\, b \,\right].$$

The objective is strictly concave in b on $[0, \infty)$ and effort is constrained to $b \geq 0$, therefore there exists a unique best response $\beta(c)$. At the edge case of $b = 0$, zero effort is optimal if and only if the right marginal payoff at zero is non-positive. Otherwise, the unique optimum is interior and satisfies the first-order condition. Accordingly, the unique optimizer $\beta(c)$ is characterized by: either $\beta(c) > 0$ and

$$V\,\mathbb{E}\left[\frac{S}{\big(\beta(c) + S\big)^2}\right] = c, \tag{FOC}$$

or $\beta(c) = 0$ and

$$\lim_{b \downarrow 0} \left\{ V\,\mathbb{E}\left[\frac{S}{(b + S)^2}\right] - c \right\} \leq 0.$$

In the symmetric i.i.d. setting, each submitter's expected payoff is strictly concave in her effort b_i, ensuring a unique best response for every realized cost c_i. Moreover, since the opponents' total effort is a strategic substitute, this fixed point is unique. Hence, the game admits a unique symmetric BNE characterized by the policy $\beta^\star(c)$ that satisfies the first-order condition (FOC).

Sybil resistance. We examine whether a submitter can gain by creating multiple pseudonymous identities, a strategy known as *Sybil attack*. Under the proportional CSF and linear costs, splitting an identity offers no advantage as long as the submitter's total effort remains fixed.

Splitting effort $\beta(c)$ across m identities $\beta(c) = \sum_{i=1}^{m} \beta_i$ yields the same winning probability and incurs the same total cost. Formally, the aggregate payoff remains

$$V\frac{\sum_{i=1}^{m} \beta_i}{S + \sum_{i=1}^{m} \beta_i} - c \sum_{i=1}^{m} \beta_i = V\frac{\beta(c)}{S + \beta(c)} - c\beta(c).$$

where S is the total effort of the other submitters. Therefore, a Sybil attack is not profitable.

7.2 Bug Discovery Dynamics

To connect effort choices to discovery dynamics, we model bug discoveries as a Poisson process whose rate is proportional to the aggregate search intensity. Let $\lambda > 0$ denote a productivity parameter capturing how efficiently collective effort translates to discoveries. Given an effort profile b, the time to the first bug discovery follows

$$T \mid b \;\sim\; \mathrm{Exp}(\lambda B), \qquad B = \sum_{i=1}^{n} b_i.$$

Therefore, the probability that by time $t > 0$ a bug was not yet discovered, and the expected discovery time are

$$\Pr[T > t \mid b] = e^{-\lambda B t}, \quad \mathbb{E}[T \mid b] = \frac{1}{\lambda B}.$$

Let $B^\star$ denote aggregate effort in the unique symmetric BNE policy $\beta(\cdot)$, then

$$T \,\big|\, \mathrm{BNE} \;\sim\; \mathrm{Exp}(\lambda\, B^\star), \qquad \mathbb{E}[T \mid \mathrm{BNE}] = \frac{1}{\lambda B^\star}.$$

Parameter effect on bug discovery time.

- *Number of submitters n.* The total effort $B^\star$ increases in n, so the expected discovery time decreases in n.
- *Prize V and allocation structure.* A larger bounty V increases equilibrium effort $B^\star$, and thus decreases the expected discovery time, $\mathbb{E}[T \mid \mathrm{BNE}] = 1/(\lambda B^\star)$. Beyond the magnitude of the bounty, its allocation also shapes incentives. If the total prize V were divided equally among the first $m \le n$ valid reports, aggregate equilibrium effort would fall. Moldovanu and Sela [30] show that, for a fixed total prize, aggregate effort is maximized under a *winner-takes-all* rule. Accordingly, DeadDrop adopts a single-winner bounty design to maximize collective search intensity and minimize expected discovery time.

7.3 How the Game Informs Protocol Design

The proposed game design focuses on (1) incentivizing participation, (2) non-cooperative competition, (3) unpredictability of the winner, and (4) utility functions proportional to invested effort. It is worth noting that the Tullock contest relies on a probabilistic procedure to define the winner of a contest and on a simultaneity assumption for players' participation. While these assumptions appear infeasible in conventional settings, the design of DeadDrop makes them effectively achievable. The confidentiality given by submitting encrypted bugs together with the obliviousness provided by OMR allows us to ensure that, up until the point when a bug receiver decides to close the contest and pay the bounty, no player can know whether and how many other players submitted a bug, thus participated in the contest. This forces every player to rely on local decisions and beliefs in order to play the game, thus submitting a bug.

8 Discussion

8.1 Excluded Threats

Here we discuss some of the threats our work does not mitigate and which we leave to future work.

Duplicate attacks. A significant limitation of our functionality is that it does not prevent an adversary from registering a duplicate contract that is semantically identical to an existing one. In such a case, a bug submitter has an incentive to submit the same bug to both contracts in order to maximize their potential reward. This leakage is particularly damaging: the adversarial party controlling the duplicate contract would also learn about the vulnerability, even though they are not the intended beneficiary.

Ultimately, in a fully public setting where anyone can register arbitrary contracts, it is impossible to prevent duplicates without introducing additional trust or centralization. Even exact bytecode-level duplicate detection cannot stop a malicious actor from redeploying the same contract at a new address. Any effective solution therefore requires incorporating an external notion of contract ownership, such as a developer registry, on-chain identity, or governance mechanism, or introducing eligibility criteria, e.g., gatekeeping the bug bounty system so that only contracts with a certain threshold of value may register. Absent such measures, one must accept duplicate attacks as an inherent limitation. We leave the design of such mechanisms to future work.

(Un-)Fair payouts. Another limitation of our functionality is that it does not enforce fairness in payout. In particular, the contract creator must be disincentivized from learning a submitted bug while withholding the corresponding reward. Ideally, once a "good submission" has been made, i.e., a novel bug that satisfies the registered condition, the developer should not be able to prevent the bounty disbursement. To address this, we plan to integrate a TEE-based fair-exchange mechanism, following the approach of Tramèr et al. [43], which ensures that the developer learns the submitted bug only if the corresponding payout is released.

Resubmission attacks. We prevent submission of bugs that do not satisfy the conditions. However, the main solution does not prevent a submitter from spamming the system by resubmitting the same bug multiple times. However, the incentive mechanism design addresses the issue from a game-theoretic approach by disincentivizing collusion as well as multiple submissions, as shown in the analysis in §7.1 under *Sybil resistance*.

8.2 Applications to Traditional Bug Bounties

Although DeadDrop is designed for blockchain and smart contracts, it can also benefit traditional (Web2) bug bounty settings. First, the lack of direct or verified contact channels is not unique to blockchains. Many Web2 open-source projects also lack reliable communication paths to reach maintainers [4, 46]. In such cases, DeadDrop's privacy-preserving reporting channel remains effective for delivering vulnerability reports without revealing which project is vulnerable. Second, traditional programs face the same challenge of expensive and time-consuming vulnerability validation [45]. The automated validation mechanism can be generalized to replay and verify bug reports in sandboxed environments, significantly reducing developers' manual verification effort.

8.3 Ethical Considerations

In this research, the primary stakeholders include (1) smart contract developers, who seek private vulnerability reporting channels; (2) security researchers, who seek to efficiently submit discovered

vulnerabilities; and (3) the Web3 community, whose financial assets rely on smart contract security. Our ethical considerations follow the principles set out in the Menlo Report.

- **Beneficence:** We maximize societal benefit by facilitating the safe remediation of bugs in smart contracts. By resolving the lack of secure disclosure channels, DeadDrop can reduce the frequency of on-chain incidents and incentivize beneficial reporting over black-market exploitation.
- **Respect for Persons:** Our design strictly prioritizes the privacy and autonomy of all participants via OMR and TEEs. DeadDrop aims to ensure that the vulnerable smart contract is not exposed to the public without consent.
- **Justice:** We promote equitable access to bug bounty rewards by removing centralized gatekeepers. Unlike traditional platforms that rely on subjective human triage, DeadDrop utilizes a TEE-based validator to judge submissions strictly on technical merit, ensuring that rewards are allocated based solely on the submission validity, free from the influence of human bias.
- **Respect for Law and Public Interest:** DeadDrop aligns with public interest by operating strictly as a receiver-initiated tool. Developers must explicitly "opt-in" to receive reports, ensuring the platform serves as a collaborative security instrument rather than a vector for unsolicited extortion or harassment.

9 Related Work

Bug bounty designs. Because different systems impose distinct requirements on bug-bounty mechanisms, designing them effectively remains a central challenge in vulnerability disclosure. Tramèr et al. combine TEEs and smart contracts to achieve fair exchange, ensuring that the submitter is paid only upon delivering a valid exploit, while the receiver learns the exploit only after payment [43]. Breidenbach et al. propose the Hydra framework for the fair exchange of smart-contract vulnerabilities [7]. In particular, Hydra protects submitters from mempool front-running of their reports through a commitment scheme but cannot preserve the privacy of vulnerable contracts, as bug reports are published on-chain. In contrast, DeadDrop employs the OMR protocol to protect the privacy of vulnerable contracts, while its encrypted bug reports inherently prevent external observation and front-running. Chen et al. propose providing a "basic reward" to bug bounty hunters who fail to find a bug but can demonstrate substantial verification effort on the program [10]. Our bug bounty mechanism pays only the first valid submitter to prevent Sybil and collusive manipulation. Beyond cryptographic and mechanism-design approaches, prior empirical and user studies of existing bug bounties have examined disclosure behaviors, offering guidance for future designs [1, 3, 14, 48].

Existing bug bounty platforms for smart contracts. Existing bug bounty ecosystems for smart contracts rely on a trusted human validation performed by humans, with the primary operational difference lying in *how* bugs are validated. Centralized platforms (e.g., HackerOne, Bugcrowd, Immunefi, and HackenProof [8, 17, 19, 21]) employ professional *in-house traige teams* to validate individual reports. While optional, this service is utilized in virtually all programs to handle high submission volumes and filter invalid or duplicate reports. In contrast, contest-based platforms (e.g., Code4rena and Hats Finance [12, 20]) run time-boxed audits where a large volume of findings arrive simultaneously and are validated by a *review committee* of curated community experts. In these crowdsourced audit models, this expert review is a core component of the workflow. All existing platforms ultimately rely on trusted human validation, through either in-house or committee-based, whereas DeadDrop supports end-to-end private, oblivious bug reporting without relying on such trusted party.

Incentives analysis for bug bounties. Prior work has examined the economic and strategic foundations of bug-bounty programs using game-theoretic models. Sarne and Lepioshkin [39] study multi-prize allocation for general crowdsourcing contests where participants strategically decide whether to enter or abstain. Gersbach et al. [15] extend this framework to bug-bounty systems, modeling vulnerability discovery as an incomplete-information game in which each researcher has a private search cost. However, they assume all searchers invest efforts to reach the same probability of finding the bug, and the searchers decide to either participate or not, given this effort level. Zhou et al. [50] model bug bounty as a Tullock contest between the company and searchers competing to find vulnerabilities. In our model, searchers compete against each other. Each searcher chooses their desired effort level, and the probability of finding the bug is derived from it. Finally, Zhou et al. [51] propose an incentive mechanism to encourage attackers to disclose leaked signing keys obtained through side-channel attacks on TEE-based cryptocurrency wallets.

10 Conclusion

In this paper, we introduce DeadDrop, the first system that offers a practical and privacy-preserving channel for disclosing smart contract vulnerabilities. By combining oblivious message retrieval (OMR) with a TEE-based validation module, DeadDrop allows submitters to report vulnerabilities without revealing the target contract or spamming messages to all developers, while developers can retrieve valid reports efficiently without exposing their identity. Our design includes a modification to the OMR protocol to support long reports and introduces a specification language to formalize bug submission. We implement a prototype of DeadDrop and evaluate its performance, showing that oblivious bug reporting can be carried out with low overhead, with an amortized processing time of about 3 ms per submission. The incentive analysis shows that DeadDrop's incentive mechanism can sustain participation and reduce expected discovery time while remaining resistant to Sybil and collusive strategies. Taken together, these results show that private disclosure is feasible for decentralized systems and open opportunities for strengthening the interaction between researchers and smart contract developers.

Author Contributions

- Mariarosaria and Jasleen contributed to the design, analysis, and implementation of the cryptographic protocol.
- Stephanie and Silei designed the bug validation specification language.
- Stephanie, Silei, and Yoshi contributed to the design and implementation of the TEE-based bug validator.
- Yoshi implemented and evaluated the OMR-based submission and retrieval mechanism.

- Marwa and Mariarosaria contributed to the incentive analysis.
- Sen contributed to the TEE-based bug validator.
- Sen and Fan identified the problem of responsible disclosure for smart contracts during their prior work. Fan sketched the initial solution ideas.
- All authors contributed to the writing of the paper.

Acknowledgments

Sen Yang and Fan Zhang are partially supported by a subaward from NSF Award #2207214. We thank Yunhao Wang for providing the OMR library implementation and for her guidance on the integration. We thank the Dstack community team for their guidance on the functionality of the Dstack SDK. We thank Jonas Chen, Yavor Litchev, and Cynthia Wang for contributing to the initial prototype at the IC3 bootcamp 2025.

References

[1] Omer Akgul, Taha Eghtesad, Amit Elazari, Omprakash Gnawali, Jens Grossklags, Michelle L Mazurek, Daniel Votipka, and Aron Laszka. 2023. Bug Hunters' Perspectives on the Challenges and Benefits of the Bug Bounty Ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2275–2291.

[2] AMD. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* 53, 2020 (2020), 1450–1465.

[3] Jessy Ayala, Steven Ngo, and Joshua Garcia. 2025. A Deep Dive into How Open-Source Project Maintainers Review and Resolve Bug Bounty Reports . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 522–538. https://doi.org/10.1109/SP61157.2025.00063

[4] Jessy Ayala, Yu-Jye Tung, and Joshua Garcia. 2025. A Mixed-Methods Study of Open-Source Software Maintainers On Vulnerability Management and Platform Security Features. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, Seattle, WA, 2105–2124.

[5] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2024. Bitcoin as a transaction ledger: A composable treatment. *Journal of Cryptology* 37, 2 (2024), 18.

[6] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. 2014. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. Association for Computing Machinery, New York, NY, USA, 15–29.

[7] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. 2018. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1335–1352.

[8] Bugcrowd. 2025. Vulnerability Triage & Validation. https://www.bugcrowd.com/products/platform/triage/. Accessed: 2025-11-14.

[9] CertiK. 2024. Hack3d: The Web3 Security Quarterly Report — Q2 + H1 2024. https://www.certik.com/resources/blog/hack3d-the-web3-security-quarterly-report-q2-h1-2024. Published: 2024-07-03; Accessed: 2025-10-29.

[10] Hongbo Chen, Quan Zhou, Sen Yang, Sixuan Dang, Xing Han, Danfeng Zhang, Fan Zhang, and XiaoFeng Wang. 2025. Agora: Trust Less and Open More in Verification for Confidential Computing. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 1372–1399.

[11] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel tdx demystified: A top-down approach. *Comput. Surveys* 56, 9 (2024), 1–33.

[12] Code4rena. 2025. Judges | Code4rena Docs. https://docs.code4rena.com/roles/judges. Accessed: 2025-11-14.

[13] Dstack-TEE. 2025. dstack: Secure TEE-based computing framework. https://github.com/Dstack-TEE/dstack.

[14] Matthew Finifter, Devdatta Akhawe, and David Wagner. 2013. An Empirical Study of Vulnerability Rewards Programs. In *22nd USENIX security symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 273–288.

[15] Hans Gersbach, Akaki Mamageishvili, and Fikri Pitsuwan. 2023. Decentralized Attack Search and the Design of Bug Bounty Schemes. arXiv:2304.00077 [econ.TH] https://arxiv.org/abs/2304.00077

[16] Google Inc. 2025. Google Bug Hunters. https://bughunters.google.com/.

[17] HackenProof. 2025. HackenProof — Bug Bounty Platform for Crypto. https://hackenproof.com/. Accessed: 2025-11-14.

[18] HackerOne. 2014. Introducing Reputation. https://www.hackerone.com/blog/introducing-reputation. Accessed: 2025-09-25.

[19] HackerOne. 2025. Bug Bounty Programs | HackerOne. https://hackerone.com/bug-bounty-programs.

[20] Hats Finance Team. 2025. Welcome to Hats.finance. https://docs.hats.finance/. Accessed: 2025-11-14.

[21] Immunefi. 2025. Managed Triage Service. https://immunefi.com/managed-triage/. Accessed: 2025-11-14.

[22] Jack Grigg. 2025. age: A simple, modern, and secure file encryption library. https://crates.io/crates/age.

[23] Zeyu Liu, Katerina Sotiraki, Eran Tromer, and Yunhao Wang. 2025. Snake-eye Resistant PKE from LWE for Oblivious Message Retrieval and Robust Encryption. In *Advances in Cryptology - EUROCRYPT 2025*. IACR, Springer Nature Switzerland, Cham, 126–156.

[24] Zeyu Liu and Eran Tromer. 2022. Oblivious Message Retrieval. In *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13507)*, Yevgeniy Dodis and Thomas Shrimpton (Eds.). Springer, 753–783. https://doi.org/10.1007/978-3-031-15802-5_26

[25] Zeyu Liu, Eran Tromer, and Yunhao Wang. 2024. PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation. In *USENIX Security Symposium*. USENIX Association.

[26] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 783–800. https://www.usenix.org/conference/usenixsecurity19/presentation/matetic

[27] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, Springer-Verlag, Berlin, Heidelberg, 369–378.

[28] Meta Platforms, Inc. 2025. Meta Bug Bounty. https://bugbounty.meta.com/. Accessed: 2025-09-25.

[29] Microsoft. 2025. Microsoft Bounty Programs | MSRC. https://www.microsoft.com/en-us/msrc/bounty.

[30] Benny Moldovanu and Aner Sela. 2001. The optimal allocation of prizes in contests. *American Economic Review* 91, 3 (2001), 542–558.

[31] Rafael Pass, Elaine Shi, and Florian Tramer. 2017. Formal abstractions for attested execution secure processors. In *Advances in Cryptology – EUROCRYPT 2017*. Springer, Springer International Publishing, Cham, 260–289.

[32] Phala Network. 2025. Phala Documentation. https://docs.phala.network/.

[33] Phala Network. 2025. TEE Attestation Explorer. https://proof.t16z.com/. https://proof.t16z.com/

[34] Kaihua Qin, Stefanos Chaliasos, Liyi Zhou, Benjamin Livshits, Dawn Song, and Arthur Gervais. 2023. The blockchain imitation game. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3961–3978.

[35] RustCrypto Developers. 2025. block-modes: Generic implementation of block cipher modes of operation, including CBC and ECB modes. https://crates.io/crates/block-modes.

[36] RustCrypto Developers. 2025. block-padding: Padding and unpadding of messages divided into blocks. https://crates.io/crates/block-padding.

[37] RustCrypto Developers. 2025. sha2: Pure Rust implementation of the SHA-2 hash function family including SHA-224, SHA-256, SHA-384, and SHA-512. https://crates.io/crates/sha2.

[38] RustCrypto Developers. 2025. aes: Pure Rust implementation of the Advanced Encryption Standard (a.k.a. Rijndael). https://crates.io/crates/aes.

[39] David Sarne and Michael Lepioshkin. 2017. Effective Prize Structure for Simple Crowdsourcing Contests with Participation Costs. *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing* 5, 1 (Sep. 2017), 167–176. https://doi.org/10.1609/hcomp.v5i1.13305

[40] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[41] Foundry Team. 2025. Anvil — Overview. https://getfoundry.sh/anvil/overview/. Accessed: 2025-10-22.

[42] Foundry Team. 2025. Forge — Overview. https://getfoundry.sh/forge/overview. Accessed: 2025-10-22.

[43] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, IEEE Computer Society, Los Alamitos, CA, USA, 19–34.

[44] Gordon Tullock. 1975. On The Efficient Organization Of Trials. *Kyklos* 28, 4 (November 1975), 745–762. https://doi.org/10.1111/j.1467-6435.1975.tb02172.x

[45] Nick Wellnhofer. 2025. Triaging security issues reported by third parties (#913). https://gitlab.gnome.org/GNOME/libxml2/-/issues/913. Accessed 2025-10-26.

[46] WinonaRyder. 2020. Ask HN: How do you responsibly report security bugs to open-source projects? https://news.ycombinator.com/item?id=21920142. Accessed: 2025-10-25.

[47] Sen Yang, Kaihua Qin, Aviv Yaish, and Fan Zhang. 2025. Insecurity Through Obscurity: Veiled Vulnerabilities in Closed-Source Contracts. https://doi.org/10.48550/arXiv.2504.13398 arXiv:2504.13398 [cs].
[48] Mingyi Zhao, Jens Grossklags, and Peng Liu. 2015. An empirical study of web vulnerability discovery ecosystems. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. Association for Computing Machinery, New York, NY, USA, 1105–1117.
[49] Mingyi Zhao, Aron Laszka, and Jens Grossklags. 2017. Devising Effective Policies for Bug-Bounty Platforms and Security Vulnerability Discovery. *Journal of Information Policy* 7 (2017), 372–418. https://doi.org/10.5325/jinfopoli.7.2017.0372
[50] Jiali Zhou and Kai-Lung Hui. 2020. Sleeping with the Enemy: An Economic and Security Analysis of Bug Bounty Programs. *HKUST Business School Research Paper* 2021-038 (2020).
[51] Lulu Zhou, Zeyu Liu, Fan Zhang, and Michael K. Reiter. 2024. CrudiTEE: A Stick-And-Carrot Approach to Building Trustworthy Cryptocurrency Wallets with TEEs. In *6th Conference on Advances in Financial Technologies (AFT 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 316)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:25. https://doi.org/10.4230/LIPIcs.AFT.2024.16
[52] Shunfan Zhou, Kevin Wang, and Hang Yin. 2025. Dstack: A Zero Trust Framework for Confidential Containers. *arXiv preprint arXiv:2509.11555* (2025).

A Formalizing Security

A.1 Helper Functionalities

The TEE ideal functionality in works such as [31]is more general. In our protocol, we simply wish to use the TEE to generate an *attestation* or *proof* that an invariant in a set of committed invariants (checked using a Merkle Tree) is broken by a given bug. Thus, we omit the general requirements of such a functionality and present a targeted version here.

A.2 Real-World Protocol In the Hybrid UC Model

In this section, we present an abstracted description for our protocol from §4, denoted Π_{DeadDrop} using $\mathcal{F}_{\mathsf{BB}}$, $\mathcal{F}_{\mathsf{OMR}}$ and $\mathcal{F}_{\mathsf{TEE}}$. Our real-world protocol instantiates each component and we defer the reader to other work [5, 24, 31]for proofs of the sub-components.

Initialization: To initialize, the server sets up functionalities $\mathcal{F}_{\mathsf{BB}}$, $\mathcal{F}_{\mathsf{OMR}}$ and $\mathcal{F}_{\mathsf{TEE}}$ by calling $(sid, \textsc{Initialize})$ on each of them.

Registering a contract: To register a contract, a receiver R, submits its invariants to $\mathcal{F}_{\mathsf{BB}}$. Note that we are implicitly assuming that the receiver is represented by the identity of the smart contract which belongs to it (including a proof of being its developer), and that the condition it posts comes with an encryption public key pk_{R}. Thus, to register its contract with invariants cond, the receiver sends the message $(ssid, \textsc{SubmitMsg}, \mathsf{cond})$ to $\mathcal{F}_{\mathsf{BB}}$.

Retrieving contract information: To retrieve contract information (i.e., the encryption public key pk_{R} and the invariant cond), the submitter, sub retrieves all the registered contracts from the bug bounty system by sending the message $(ssid, \textsc{ReceiveAll})$ to $\mathcal{F}_{\mathsf{BB}}$. sub will need the invariants for all of the contracts registered to the bug bounty program to be able to show that the invariant it breaks is actually one of the registered ones. At the end of this step, sub will hold an ordered (by *ssid*) set Conds of (R, cond) tuples.

Submitting a bug: In order to submit a bug, a submitter sub creates a Merkle tree over Conds and retrieves the root hash of this tree, denoted com. Suppose (R, cond) is the actual invariant sub has a bug report Report for. Then, it creates a Merkle membership proof π for (R, cond) with respect to com and sends the message $(ssid, \textsc{Prove}, \mathsf{com}, (\mathsf{R}, \mathsf{cond}), c, \mathsf{Report}, \pi)$ to $\mathcal{F}_{\mathsf{TEE}}$ and $(ssid, \textsc{SubmitMsg}, \mathsf{R}, c)$ to $\mathcal{F}_{\mathsf{OMR}}$.

We assume some synchrony here, and for its part, S, when prompted by $\mathcal{F}_{\mathsf{TEE}}$, also sends $(ssid', \textsc{ReceiveAll})$ to $\mathcal{F}_{\mathsf{BB}}$, and uses this response to compute a commitment com′. Then, it verifies the submission for this *ssid*, by sending the message $(ssid, \mathsf{Vf}, \mathsf{com}')$ to $\mathcal{F}_{\mathsf{TEE}}$. If the response from $\mathcal{F}_{\mathsf{TEE}}$ is of the form $(ssid, \mathsf{sub}, c, 1, 1)$, S stores this message and moves forward, otherwise it aborts. Upon message $(ssid, \textsc{SubmitMsg}, \mathsf{sub}, c)$ from $\mathcal{F}_{\mathsf{OMR}}$, assuming this matches the corresponding stored $\mathcal{F}_{\mathsf{TEE}}$ message, S responds with $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub})$ and clears the stored messages.

Receiving a bug: In order to retrieve bugs destined for it, a receiver sends a message $(ssid, \textsc{ReceiveMsg})$ to $\mathcal{F}_{\mathsf{OMR}}$. Upon receiving a response of the form $(ssid, \mathsf{FoundSet})$, it parses $\mathsf{FoundSet} := \{(ssid_j, \mathsf{sub}_j, \mathsf{R}, c_j)\}$, and computes $\mathsf{sk}_j = \mathsf{Dec}(\mathsf{sk}_{\mathsf{R}}, c_j[0])$ and $\mathsf{Report}_j = \mathsf{AES.Dec}(\mathsf{sk}_j, c_j[1])$.

A.3 The Ideal Functionality for Oblivious Bug Reporting

Mapping security to the ideal functionality. Each of these threat-model requirements corresponds directly to features of $\mathcal{F}_{ob}$ in Figure 8. The register and update condition procedures allow developers (receivers) to specify invariants for their contracts. The retrieve contract info interface ensures that submitters can obtain the conditions they need to test, while the submit bug routine enforces that validity is checked against the developer's registered condition without disclosing either the contract or the bug content to the server. Finally, the retrieve bug procedure allows only the appropriate receiver to learn about valid submissions for their contract, ensuring that developers but not outside parties learn of existing vulnerabilities. At the same time, the functionality makes clear what information is inevitably leaked: the server (and any network adversary observing traffic) learns the timing, frequency, and sizes of messages exchanged, as well as which parties are sending or retrieving messages at a given time. The server also learns whether a submitted bug is valid with respect to *some* contract's invariants, however, it does not learn the actual bug contents or which specific contract/invariants this bug pertains to.

The design further assumes that the server is trusted for availability: every operation in the functionality requires an acknowledgment from the server (an "ok" message) before the action is considered complete. This trust assumption implies that if the server withholds responses, it can prevent progress. Consequently, we place censorship resistance out of scope. Our functionality ensures confidentiality and correctness of delivery, but does not protect against an adversarial server that simply refuses to forward messages.

B Security Proof Sketch

Here we provide a proof sketch for Theorem 1. Our threat model considers the following parties with their corresponding threat assumptions:

- Receiver: The receiver is considered to be honest throughout the protocol, and thus we will not be simulating this.
- Submitter: The submitter may be malicious.
- Server: The server is assumed to be honest-but-curious.

Now, we consider two cases of corruption: corrupted submitter and corrupted server, and provide a simulator for each case. Since

Functionality $\mathcal{F}_{\mathsf{OMR}}$ for oblivious message retrieval.

Parties: A single server S, a set of receivers R_1, ..., R_k, a set of submitters sub_1, ..., sub_m and a network adversary $\mathcal{A}$.
Public Parameters: A security parameter λ.
Initialization: On message $(sid, \textsc{Initialize})$ from S, initialize a buffer buf and a message set MsgSet.
Send message:

- Upon message $(ssid, \textsc{SubmitMsg}, \mathsf{R}_i, \mathsf{msg})$ from sub_j, add $(ssid, \textsc{SubmitMsg}, \mathsf{sub}_j, \mathsf{R}_i, \mathsf{msg})$ to buf and send message $(ssid, \textsc{SubmitMsg}, \mathsf{sub}_j, \mathsf{msg})$ to S.
- Upon message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_j)$ from S, if there exists a record of the form $r = (ssid, \textsc{SubmitMsg}, \mathsf{sub}_j, \mathsf{R}_i, \mathsf{msg})$ in buf, add $(ssid, \mathsf{sub}_j, \mathsf{R}_i, \mathsf{msg})$ to MsgSet. Remove r from buf if it exists. Send message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_j, \mathsf{msg})$ to $\mathcal{A}$.

Receive message

- Upon message $(ssid, \textsc{ReceiveMsg})$ from R_i, add $(ssid, \textsc{ReceiveMsg}, \mathsf{R}_i)$ to buf and send message $(ssid, \textsc{ReceiveMsg})$ to $\mathcal{A}$.
- Upon message $(ssid, \textsc{ReceiveMsgOk})$ from $\mathcal{A}$, if there exists a record $(ssid, \textsc{ReceiveMsg}, \mathsf{R}_i)$ in the buffer do the following. Let FoundSet be the set of all records of the form $r = (\cdot, \cdot, \mathsf{R}_i, \mathsf{msg})$ in MsgSet (i.e. R_i is the receiving party). Remove $(ssid, \textsc{ReceiveMsg}, \mathsf{R}_i)$ from buf and send $(ssid, \textsc{Received}, \mathsf{FoundSet})$ to R_i. Send $(ssid, \textsc{Received}, |\mathsf{FoundSet}|)$ to $\mathcal{A}$.

Figure 5: The oblivious message retrieval ideal functionality. Note that in general OMR, using a bulletin-board, any submitter may be allowed to post or there may exist a set of authorized parties who facilitate message posting, but to make notation easier for our proof, we just assume the server is the only party that can authorize posting of messages.

Functionality $\mathcal{F}_{\mathsf{BB}}$ for a bulletin-board.

Parties: A set of parties $P = \{P_1, ..., P_n\}$, a setup or "server" party S that just initializes the bulletin-board, and a network adversary $\mathcal{A}$.
Initialization: On message $(sid, \textsc{Initialize})$ from S, initialize a buffer buf and a message set MsgSet.
Post message:

- Upon message $(ssid, \textsc{SubmitMsg}, \mathsf{msg}_i)$ from P_i, if this is the first such message from P_i, add $(ssid, P_i, \mathsf{msg}_i)$ to MsgSet. Send message $(ssid, \textsc{SubmitMsg}, P_i, \mathsf{msg}_i)$ to $\mathcal{A}$ and S.

Read posts

- Upon message $(ssid, \textsc{ReceiveAll})$ from a party $\mathsf{P} \in P \cup \{\mathsf{S}, \mathcal{A}\}$, send message $(ssid, \textsc{ReceiveAll}, \mathsf{P})$ to $\mathcal{A}$ and, send message $(ssid, \textsc{ReceiveAll}, \mathsf{MsgSet})$ to P.

Figure 6: The bulletin-board ideal functionality.

Functionality $\mathcal{F}_{\mathsf{TEE}}$ for a TEE that attests that the invariant cond **for some registered contract is broken by a particular bug.**

Parties: A set of parties $P = \{P_1, ..., P_n\}$, a special verifying party, denoted S, and a network adversary $\mathcal{A}$.
Initialization: On message $(sid, \textsc{Initialize})$ from S, initialize empty sets Stmts and Fails.
Prove: On message $(ssid, \textsc{Prove}, \mathsf{com}, (\mathsf{R}, \mathsf{cond}), c, \mathsf{sk}, \mathsf{Report}, \pi)$ from party P_i, check the following:

- $\mathsf{Verify}_{\mathcal{M}}(\mathsf{com}, (\mathsf{R}, \mathsf{cond}), \pi)$ outputs 1, set $b_{\mathsf{Integrity}} = 1$, else set $b_{\mathsf{Integrity}} = 0$.
- $c = (\mathsf{Enc}(\mathsf{pk}_{\mathsf{R}}, \mathsf{sk}), \mathsf{AES.Enc}(\mathsf{sk}, \mathsf{Report}))$. If this test fails, set $b_{\mathsf{Integrity}} = 0$.
- Report violates the invariant cond.

If all checks verify, add $(ssid, \mathsf{com}, (\mathsf{R}, \mathsf{cond}), \mathsf{Report}, c, b_{\mathsf{Integrity}}, P_i)$ to Stmts. Else, add $(ssid, \mathsf{com}, (\mathsf{R}, \mathsf{cond}), \mathsf{Report}, c, b_{\mathsf{Integrity}}, P_i)$ to Fails. Send message $(ssid, \textsc{Prove}, P_i)$ to S.
Verify: Upon message $(ssid, \mathsf{Vf}, \mathsf{com})$ from S, if there exists an entry: $(ssid, \mathsf{com}^*, (\mathsf{R}, \mathsf{cond}), \mathsf{Report}, c, b_{\mathsf{Integrity}}, P_i)$ in Stmts, let $b_{\mathsf{SAT}} = 1$, else, if the record is in Fails, let $b_{\mathsf{SAT}} = 0$. If the record did exist in either of Fails or Stmts, let $b_{\mathsf{cond}} = 1 \wedge b_{\mathsf{Integrity}}$ if $\mathsf{com} = \mathsf{com}^*$, else, set $b_{\mathsf{cond}} = 0$. If b_{cond} and b_{SAT} exist, send message $(ssid, P_i, c, b_{\mathsf{cond}}, b_{\mathsf{SAT}})$ to S.

Figure 7: The TEE ideal functionality.

Functionality $\mathcal{F}_{ob}$ for oblivious submission and recovery of bugs

Parties: A single server S, a set of receivers R_1, ..., R_k and a set of submitters sub_1, ..., sub_m.
Public Parameters: A security parameter λ.
Initialization: On message (Initialize) from S, initialize an empty buffer buf, and empty sets $\mathcal{B}$ (entered bugs) and an empty set, Conds (registered conditions for contracts). Send message (sid, ServerInit) to $\mathcal{A}$. Until all three sets have been initialized, ignore any other messages.
Register contract:

- Upon message ($ssid$, Register, cond) from receiver R_i, if this is a the first such message from R_i, add add ($ssid$, R_i, cond_i) to Conds and send message ($ssid$, Register, R_i, cond_i) to S and $\mathcal{A}$.

Retrieve contract info:

- Upon message ($ssid$, RetrieveInfo) from a party P, send message ($ssid$, RetrieveInfo, P) to $\mathcal{A}$ and ($ssid$, Conds) to P.

Submit bug:

- Upon message ($ssid$, SubmitBug, R_i, cond_i^*, Report, $b_{\mathsf{Integrity}}$) from sub_j, add ($ssid$, SubmitBug, sub_j, R_i, cond_i^*, Report, $b_{\mathsf{Integrity}}$) to buf and send message ($ssid$, SubmitBug, sub_j) to S.
- Upon message ($ssid$, CheckSubmission, sub_j) from S, if buf contains a record r of the form ($ssid$, SubmitBug, sub_j, R_i, cond_i^*, Report, $b_{\mathsf{Integrity}}$), if there exists a condition of the form ($\cdot$, R_i, cond_i) in Conds, such that $\mathsf{cond}_i = \mathsf{cond}_i^*$, set $b_{\mathsf{cond}} = 1 \wedge b_{\mathsf{Integrity}}$. Set $b_{\mathsf{cond}} = 0$ and $b_{\mathsf{SAT}} = \mathsf{cond}_i^*(\text{Report})$. Send message ($ssid$, CheckSubmission, sub_j, b_{cond}, b_{SAT}) to S.
- Upon message ($ssid$, SubmitOk, sub_j) from S, if there exists a record of the form r = ($ssid$, CheckedBug, sub_j, R_i, Report) in buf, add ($ssid$, sub_i, R_j, Report) to $\mathcal{B}$. Remove r from buf if it exists. Send ($ssid$, SubmitMsgOk, sub_j) to $\mathcal{A}$.

Receive bug

- Upon message ($ssid$, RetrieveBug, Report) from R_i, if there exist bugs b_1, ..., b_n of the form b_ℓ = ($\cdot$, sub_{j_ℓ}, R_i, Report) in $\mathcal{B}$, send message ($ssid$, Retrieved, $\{b_1, ..., b_n\}$) to R_i and ($ssid$, Retrieved, n) to $\mathcal{A}$. In any case, delete r from buf.

Figure 8: The oblivious bug submission functionality.

the receiver is always considered to be honest, we do not consider the case of a corrupted receiver. Secondly, since the model only allows honest-but-curious corruption for the server, the server would never collude with a corrupted submitter, since this is outside the specifications of its behaviour (recall, an honest-but-curious party follows the specification of the protocol). Thus, we need not consider the case where both the server and submitter are corrupted. We do need to consider the case where multiple submitters are corrupted, but the simulator in this case is just the same as the simulator in the single corrupt submitter case, working with corresponding messages for each submitter corrupted by the adversary.

Note that we assume the network implements secure channels between all honest parties and the ideal functionality.

Corrupted Submitter. Given a real-world adversary $\mathcal{A}$, who corrupts submitter sub_j, the simulator Sim proceeds as follows:

- **Initialization:** Sim internally runs a copy of $\mathcal{A}$. On message (sid, ServerInit) from $\mathcal{F}_{ob}$, initialize an internal copy of $\mathcal{F}_{\mathsf{BB}}$ and a copy of $\mathcal{F}_{\mathsf{OMR}}$. Sim forwards to its copy of $\mathcal{A}$ any messages from the environment meant for $\mathcal{A}$.
- **Register contract:** Upon message ($ssid$, Register, R_i, cond_i) from $\mathcal{F}_{ob}$, meant for the adversary, Sim sends message ($ssid$, SubmitMsg, i, cond_i) to its internal copy of $\mathcal{F}_{\mathsf{BB}}$ as if it came from R_i. It forwards any message from $\mathcal{F}_{\mathsf{BB}}$ to $\mathcal{A}$.
- **Retrieve contract info:** Upon message ($ssid$, ReceiveAll) from $\mathcal{A}$, meant for $\mathcal{F}_{\mathsf{BB}}$, send message ($ssid$, RetrieveInfo) to $\mathcal{F}_{ob}$. Upon message ($ssid$, Conds) from $\mathcal{F}_{ob}$, send message ($ssid$, ReceiveAll, sub_j) to $\mathcal{A}$ as if it came to the network adversary from $\mathcal{F}_{\mathsf{BB}}$ and also the message ($ssid$, ReceiveAll, Conds) as if $\mathcal{F}_{\mathsf{BB}}$ sent it for sub_j. Locally store ($ssid$, Conds), compute and store $\mathsf{com}_{\mathsf{Conds}}$ as the root of a Merkle tree of the (R, cond) tuples in Conds.
- **Submit bug:**
  - Upon message ($ssid$, Prove, com, (R, cond), c, sk, Report, π) meant for $\mathcal{F}_{\mathsf{TEE}}$, from $\mathcal{A}$, if $\mathsf{com} \neq \mathsf{com}_{\mathsf{Conds}}$ or, if $\mathsf{Verify}_{\mathcal{M}}(\mathsf{com}, (\mathsf{R}, \mathsf{cond}), \pi)$ outputs 0, or, if $c \neq (\mathsf{Enc}(\mathsf{pk}_\mathsf{R}, \mathsf{sk}), \mathsf{AES.Enc}(\mathsf{sk}, \text{Report}))$, set $b_{\mathsf{Integrity}} = 0$, else, set $b_{\mathsf{Integrity}} = 1$. Send message ($ssid$, SubmitBug, R, cond, Report, $b_{\mathsf{Integrity}}$) to $\mathcal{F}_{ob}$.
  - Upon message ($ssid$, SubmitMsgOk, j) from $\mathcal{F}_{ob}$ send message ($ssid$, SubmitMsg, R, c) to $\mathcal{F}_{\mathsf{OMR}}$ as sub_j. Upon message ($ssid$, SubmitMsg, sub_j, c), meant for S from $\mathcal{F}_{\mathsf{OMR}}$, respond with message ($ssid$, SubmitMsgOk, sub_j, c) to $\mathcal{F}_{\mathsf{OMR}}$, as if it came from S. Finally, forward the message ($ssid$, SubmitMsg, sub_j, c) meant for $\mathcal{A}$ from $\mathcal{F}_{\mathsf{OMR}}$.
  - Ignore message ($ssid$, SubmitMsgOk, j) from $\mathcal{F}_{ob}$ meant for $\mathcal{A}$.
- **Receive bug:** Upon message ($ssid$, Retrieved, n) from $\mathcal{F}_{ob}$, send message ($ssid$, Received, n) to $\mathcal{A}$, as if it were sent from $\mathcal{F}_{\mathsf{OMR}}$.

Note that in case multiple submitters are corrupted, sub runs each of them internally and behaves as above.

Corrupted Server. Now, we build a simulator for a real-world adversary $\mathcal{A}$, who is semi-honest, also known as honest-but-curious, and corrupts S.

- **Initialization:** Again, the simulator, Sim will run internal copies of $\mathcal{F}_{\mathsf{BB}}$ and $\mathcal{F}_{\mathsf{OMR}}$, it will not run a copy of $\mathcal{F}_{\mathsf{TEE}}$ internally but simply impersonate it to $\mathcal{A}$ when needed. When $\mathcal{A}$ sends initialization messages meant for each ideal functionality, Sim sends message $(sid, \textsc{Initialize})$ to $\mathcal{F}_{ob}$, as if it came from S.
- **Register contract:** Upon message $(ssid, \textsc{Register}, \mathsf{R}, \mathsf{cond})$ from $\mathcal{F}_{ob}$ (meant for $\mathcal{A}$ and S), send message $(ssid, \textsc{SubmitMsg}, \mathsf{cond})$ to the internal copy of $\mathcal{F}_{\mathsf{BB}}$, as if it came from R. Forward the messages emitted by $\mathcal{F}_{\mathsf{BB}}$ after this posting to $\mathcal{A}$.
- **Retrieve contract info:** On message $(ssid, \textsc{RetrieveInfo}, \mathsf{sub}_j)$ from $\mathcal{F}_{ob}$, send message $(ssid, \textsc{ReceiveAll})$ to $\mathcal{F}_{\mathsf{BB}}$ as if it came from sub_j. On response $(ssid, \textsc{ReceiveAll}, \mathsf{sub}_j)$, pass this message to $\mathcal{A}$ as if it came from $\mathcal{F}_{\mathsf{BB}}$.
- **Submit bug:** On message $(ssid, \textsc{SubmitBug}, \mathsf{sub}_j)$ from $\mathcal{F}_{ob}$, send message $(ssid, \textsc{Prove}, \mathsf{sub}_i)$ to $\mathcal{A}$ as if it came from $\mathcal{F}_{\mathsf{TEE}}$. Upon message $(ssid, \textsc{ReceiveAll})$ from $\mathcal{A}$, pass this message to the internal copy of $\mathcal{F}_{\mathsf{BB}}$ and forward its response to $\mathcal{A}$. Meanwhile, Sim locally also computes a Merkle tree over the conditions to get the correct commitment com.
  Upon message $(ssid, \mathsf{Vf}, \mathsf{com}')$ from $\mathcal{A}$ meant for $\mathcal{F}_{\mathsf{TEE}}$, send message $(ssid, \textsc{CheckSubmission}, \mathsf{sub}_i)$ to $\mathcal{F}_{ob}$. Upon response, $(ssid, \textsc{CheckSubmission}, \mathsf{sub}_i, b_{\mathsf{cond}}, b_{\mathsf{SAT}})$, set $b'_{\mathsf{cond}} = b_{\mathsf{cond}} \wedge (\mathsf{com} = \mathsf{com}')$. Select a random string c of the correct length to encode a real-world bug report Report, and send $(ssid, \mathsf{sub}_i, c, b'_{\mathsf{cond}}, b_{\mathsf{SAT}})$ to $\mathcal{A}$ as if it came from $\mathcal{F}_{\mathsf{TEE}}$. Also send message $(ssid, \mathsf{sub}_i, c)$ to $\mathcal{A}$ as if it came from $\mathcal{F}_{\mathsf{OMR}}$.
  Upon message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_i)$ from $\mathcal{A}$, meant for $\mathcal{F}_{\mathsf{OMR}}$, send message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_i)$ to $\mathcal{F}_{ob}$ and on message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_i)$ from $\mathcal{F}_{ob}$, meant for the network adversary, send message $(ssid, \textsc{SubmitMsgOk}, \mathsf{sub}_i, c)$ to $\mathcal{A}$ as if it came from $\mathcal{F}_{\mathsf{OMR}}$.
- **Receive bug:** Upon message $(ssid, \textsc{Retrieved}, n)$ from $\mathcal{F}_{ob}$, send message $(ssid, \textsc{Received}, n)$ to $\mathcal{A}$, as if it came from $\mathcal{F}_{\mathsf{OMR}}$.