VeriOSS: Using the Blockchain to Foster Bug Bounty Programs

Andrea Canidio
IMT School for Advanced Studies, Lucca, Italy
INSEAD, Fontainebleau, France
andrea.canidio@imtlucca.it

Gabriele Costa
IMT School for Advanced Studies, Lucca, Italy
gabriele.costa@imtlucca.it

Letterio Galletta
IMT School for Advanced Studies, Lucca, Italy
letterio.galletta@imtlucca.it

Abstract

Nowadays software is everywhere and this is particularly true for free and open source software (FOSS). Discovering bugs in FOSS projects is of paramount importance and many *bug bounty programs* attempt to attract skilled analysts by promising rewards. Nevertheless, developing an effective bug bounty program is challenging. As a consequence, many programs fail to support an efficient and fair bug bounty market. In this paper, we present *VeriOSS*, a novel bug bounty platform. The idea behind VeriOSS is to exploit the blockchain technology to develop a fair and efficient bug bounty market. To this aim, VeriOSS combines formal guarantees and economic incentives to ensure that the bug disclosure is both reliable and convenient for the market actors.

2012 ACM Subject Classification Security and privacy → Software security engineering; Software and its engineering → Formal software verification; Security and privacy → Economics of security and privacy

Keywords and phrases Bug Bounty, Decentralized platforms, Symbolic-reverse debugging

Digital Object Identifier 10.4230/OASIcs.Tokenomics.2020.6

Funding This research is partially funded by IMT PAI project "VeriOSS".

1 Introduction

Free and open source software (FOSS) is becoming more and more popular.[1] Operating systems and applications that we use daily are often developed and maintained by consortia of partner industries and communities of developers. FOSS is even mandatory in some cases, e.g., cryptographic functions are publicly developed for transparency and revision.

Bug bounty programs are essential to attract skilled software analysts for the detection, disclosure and correction of software errors. In a bug bounty program, a *bounty issuer* (BI) offers a reward to any *bounty hunter* (BH) who discovers a bug in a piece of software. The offered reward usually depends on the typology and criticality of the bug. For instance, Google promises to pay up to 15000$ for a sandbox escape vulnerability in the Chrome web browser.[2]

Often, BI is the software developer or owner, e.g., Google in the example above. However, a bounty can be also issued for third-party software. This is the case for FOSS components

[1] https://www.forbes.com/sites/taylorarmerding/2019/01/09/the-future-of-open-source-software-more-of-everything/
[2] https://www.google.com/about/appsecurity/chrome-rewards

2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020).
Editors: Emmanuelle Anceaume, Christophe Bisière, Matthieu Bouvard, Quentin Bramas, and Catherine Casamatta; Article No. 6; pp. 6:1–6:14

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

involved in some critical systems, either open or proprietary. A prominent example of bounties for third-parties software is the *Free and Open Source Software Audit* (FOSSA) project, sponsored by the European Commission and offering bounties of up to several hundreds of thousands of euros for vulnerabilities discovered in 14 major FOSS.[3] According to the project executives, FOSSA is a response to *Heartbleed*, a severe security vulnerability that affected OpenSSL in 2014.[4]

Bug bounty programs are subject to numerous challenges. The main one is BI's lack of commitment with respect to the eligibility of bugs. Usually, a BH is expected to disclose all details of a bug to the BI who decides on the severity of the bug and therefore how much to pay. Clearly, the BI has strong incentives to "downgrade" the bug or declare it not eligible for the bounty. In this way, the BI depresses the payment to the BH who, at that point, has no more bargaining power. For example, in 2016 the majority of the security report received by Google were considered invalid.[5] This makes the bounty market inefficient and pushes BHs to look for other opportunities, such as gray and black markets.[6]

As a partial answer to this problem, mediation platforms have been created, in an effort to obtain better terms for the BHs. For instance, HackerOne[7] and Integriti[8] support ethical hackers in submitting their reports and collecting rewards. A second answer is to transform a bug into an exploit, that is an attack leveraging it. This increases the bargaining power of the BH toward the BI, and in fact some platforms exclusively focus on exploits.[9]

In this position paper, we present the design and the underlying ideas of VeriOSS, a blockchain-based platform for bug bounties. Our goal is to increase the reward for BH, so to foster more bug hunting and, consequently, decrease the appeal of grey and black markets. To do that, VeriOSS drives both BI and BH though a bug disclosure protocol. The protocol starts from the BI issuing a bug reward contract where a precise characterization of the eligible bugs is provided together with the offered reward. When a BH claims the bounty, she must provide enough information for the BI to check the eligibility without revealing the details for reproducing the bug. If the BI accepts the transaction, a remote debugging protocol is executed between the BI and the BH. At each step, BI computes a challenge that BH can only solve by continuing the debug process and revealing part of the execution trace reproducing the bug. In exchange, BI provides a commitment to pay a fraction of the total reward through a smart contract. Eventually, BI and BH either complete the protocol or interrupt it. In both cases, since BH and BI negotiate the partial rewards at each step, the protocol ensures a fair trade between the revealed information and the reward.

The rest of the paper is organized as follows. The next section introduces some preliminary notions. We present the design of VeriOSS and of its main components in Section 3. Section 4 discusses the economic incentives that drive the protocol execution. We discuss the threat model, some limitations and future extensions in Section 5. Finally, Section 6 compares our proposal with the literature, and Section 7 draws some conclusions.

---

[3] `https://juliareda.eu/fossa`
[4] `http://heartbleed.com`
[5] `https://sites.google.com/site/bughunteruniversity/behind-the-scenes/charts/2016`
[6] The activities occurring on gray and black markets are hard to document. However, Hacking Team's recently hacked emails provide a glimpse on the workings of these markets. See `https://tsyrklevich.net/2015/07/22/hacking-team-0day-market/`.
[7] `https://www.hackerone.com`
[8] `https://www.intigriti.com`
[9] For instance, Zerodium `https://zerodium.com` that offers up to 2,000,000$ for a zero-day exploit.

2 Preliminary notions

2.1 Program semantics

A program s is a finite sequence of statements $c_1, \ldots, c_k$. Statements can be of various types, e.g., assignments of values to variables or conditional branches. A computation is carried out through atomic steps. Each step has the effect of modifying the program state σ and to update the sequence of statements to be run. As usual in program semantics, the state is a finite mapping from variables (in the scope of the current statement) to values [18]. Thus, a *program configuration* is a pair $\langle \sigma, s \rangle$. A step is $\langle \sigma, s \rangle \to \langle \sigma', s' \rangle$ to denote that in the state σ the program s executes one of its statements, becomes s' and modifies the state in σ'. For brevity, we write $\langle \sigma, s \rangle \to^* \langle \sigma', s' \rangle$ as a shorthand for a finite sequence of steps $\langle \sigma, s \rangle \to \ldots \to \langle \sigma', s' \rangle$. Moreover, when a computation terminates, i.e., the destination configuration contains an empty sequence of statements, we simply write the final state as $\langle \sigma, s \rangle \to^* \sigma'$. We refer to [18] for a general presentation on the formal semantics of programming languages.

2.2 Hoare logics

The goal of program verification is to prove that a program s complies with a given specification. The specification is often defined in terms of *preconditions* and *postconditions*. Intuitively, a precondition is a property P that is assumed to hold in the initial state (from which the computation of s starts) and a postcondition is a property Q that must be guaranteed to hold in the final state (assume-guarantee reasoning). In symbols, the problem is encoded as an *Hoare triple* $\{P\}s\{Q\}$. The triple is valid if $\forall \sigma, \sigma'.\ P(\sigma) \wedge \langle \sigma, s \rangle \to^* \sigma' \Rightarrow Q(\sigma')$. The proof system used for reasoning about the validity of Hoare triples is called *Hoare logics*. We write $\models \{P\}s\{Q\}$ when there exist a proof of validity.

3 VeriOSS

In this section we introduce the main components of VeriOSS and how they interact. Briefly, VeriOSS has two goals: (i) support the honest BH in collecting a reward under the assumption of an untrusted BI; and (ii) protect BI against untrusted BHs claiming an undeserved reward. In particular, VeriOSS achieves these two goals by (i) requiring BI to provide a precise description of the eligible bugs; and (ii) driving the BH disclosure and rewarding process.

3.1 Workflow overview

The general workflow of VeriOSS is depicted in Figure 1. Initially, BI publishes a bounty on the blockchain. The bounty contains information about the type of bugs BI is interested in and the reward. When BH detects a bug that complies with the issued bounty, she can claim the reward. To do so, BH sends the initial debug information, e.g., the instruction where the bug was detected. This initial disclosure should allow BI to check the eligibility of the bug. If BI agree to continue, a disclosure loop starts. At each iteration, BI synthesizes a challenge for BH to test her knowledge of the bug trace at a specific step. If BH solves the challenge, she receives a partial reward (expressed as a fraction of the total one) and she provides information to continue the disclosure loop. Eventually, the protocol terminates when either the bug is entirely disclosed (proof completed) or one of the participants withdraws.

Below we discuss the components of VeriOSS and their requirements.

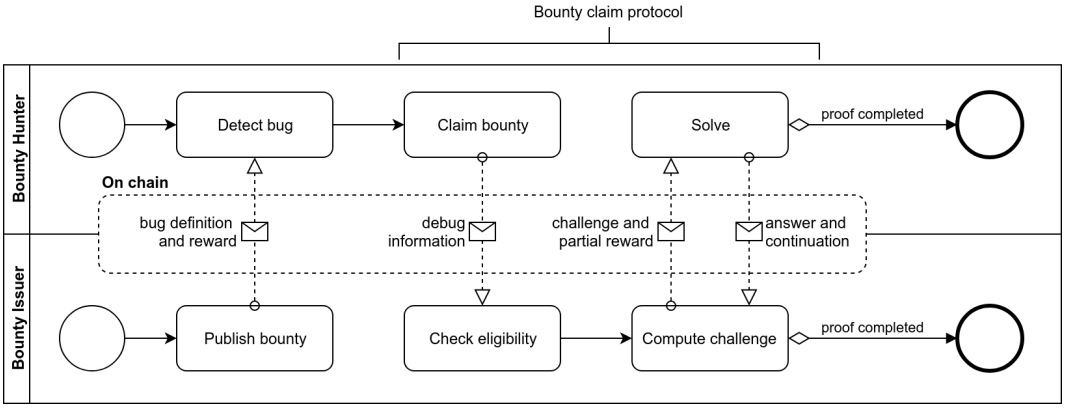

Figure 1 BPMN representation of the workflow.

```
float foo(unsigned char c) {
  int a = c+1;          //@ assert a != 0;
  float z = 255/a;      //@ assert z != 0;
  return 1.0/z;
}
```

Figure 2 A fragment of C code potentially dividing by zero.

3.2 Bug specification

When publishing a bounty, BI has to provide a rigorous description of the bugs that are eligible for the reward. Such a description contractualizes the commitment of BI to pay for a compatible bug. Some classification techniques exist to define bugs and vulnerabilities. For instance, the Common Vulnerability Scoring System[10] (CVSS) aims at describing a vulnerability and measuring its criticality. Also the Common Weaknesses Enumeration[11] (CWE) specification language is used to identify different vulnerability types. Since these approaches focus on describing the severity of vulnerabilities and exploits, they are not suitable for bug bounty programs. In fact, often the BI aims at disclosing bugs even without knowing their possible impact and severity. Moreover, since they have no formal semantics, they can hardly support an automatic validation process.

A more promising direction is to consider specification languages for contract-driven development [14]. These languages are used to define the properties of a piece of code in terms of preconditions (what must be true before the execution) and postconditions (what must be true after the execution). Moreover, they are usually provided with a formal semantics as well as tools for the automatic reasoning. Intuitively, program specifications can be adapted to define the conditions under which a bug shows up. The bug conditions can be expressed as assertions that the program violates during a bugged execution. To clarify, let us consider an example based on the ANSI C Specification Language (ACSL), used by the Frama-C framework [12].

▶ **Example 1.** Consider the C code of Figure 2. If we are interested in spotting out divisions by 0, there are two candidate instructions, i.e., the assignment to `z` and the `return` statement. In terms of properties to be satisfied, the preconditions for the two statements are $\texttt{a} \neq 0$ and

[10] `https://www.first.org/cvss/specification-document`
[11] `https://cwe.mitre.org/cwss/cwss_v1.0.1.html`

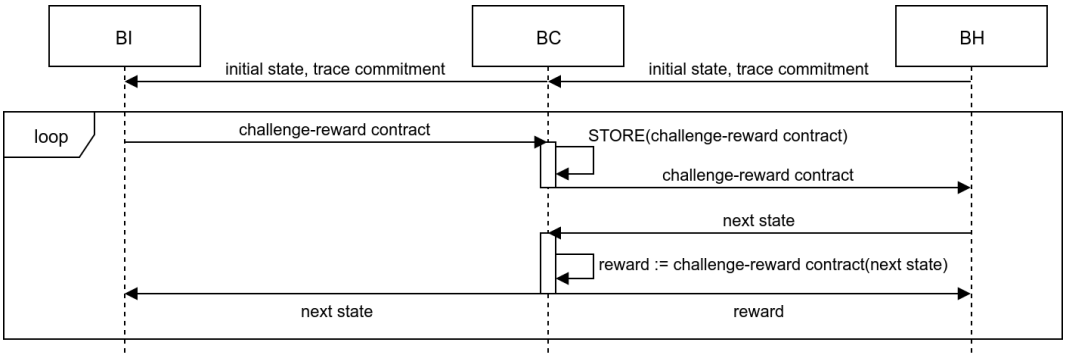

Figure 3 The P2K protocol message sequence diagram.

$z \neq 0$, respectively. In ASCL the corresponding assertions are placed right before the target instructions as in Figure 2. Here, the bug is exposed (only) when `c = 255`. As a matter of fact, due to the integer division `255/256`, 0 is assigned to `z`, so violating the second assertion.

3.3 Challenge-response interaction

By definition, the bounty claim protocol is a Proof of Knowledge (PoK) protocol, also called $\Sigma-$protocol (see [10] for further details). A PoK consists of a prover and a verifier interacting through a challenge-response process.

However, our working conditions are slightly different. The reason is that both parties need to prove something: BH must prove she knows the bug and BI must prove she is willing to pay the reward. This is an instance of a *two-party fair exchange* protocol [15] that we call *Pay-per-Knowledge* (P2K).

The main difference between a standard PoK is that the two parties play both roles, i.e., prover and verifier. Their individual goal is to acquire the other's knowledge/reward. Also, the global goal of the protocol is that the two parties only achieve their individual goals *together*. Notice that "together" does not mean simultaneously. For instance, a party could receive the other's knowledge while providing an *effective commitment* to release her own knowledge (e.g., within a certain time).

Intuitively, a way to implement P2K is to rely on a trusted third party (TTP) that mediate and drive the interaction between the two participants. However, having a TTP is a restrictive assumptions. Smart contracts can support the same kind of operation. Indeed, a smart contract can carry out a certain task when a certain condition is satisfied, e.g., someone knows the answer to a challenge. We discuss this aspect in Section 3.6.

Figure 3 shows the P2K message flow of the bounty claim protocol. The bug disclosure is based on a remote debugging process (see Section 3.4) replicating the execution of a buggy program trace. The protocol starts with BH claiming the bounty by describing the bug without disclosing it. For instance, the bug description can consist of a buggy state reached by the program at the end of the execution trace. This initial disclosure allows the BI to check the eligibility and severity of the bug, without being able to replicate it nor verify its actual existence. Contextually, the BH commits the debug trace. The commitment amounts to the hash values of the program states appearing in the trace. The trace commitment ensures that a dishonest BH can neither craft a trace not diverge from the nominal protocol execution (see Section 3.3).

The challenge-response loop proceeds as follows. BI stores a *challenge-reward* smart contract on the blockchain. Briefly, the smart contract consists of a payment, i.e., a partial reward, activated when a certain input is provided. The contract input is the answer to the challenge computed by BI. In particular, the challenge is solved by a program state from which the buggy state is reachable (in a certain number of steps). BH checks the challenge and the amount. If she agrees with the partial reward, she submits the program state. If this program state correctly solves the challenge, and at the same time is consistent with the obfuscated trace, then the BH can collect the reward. Since the blockchain is public, BI retrieves the submitted state. The loop is repeated by replacing the buggy state with the next state provided by BH. Eventually, the loop terminates when BH provides an initial state of the program or one of the parties retires from the protocol. We describe the challenge generation procedure and the smart contract implementation in Sections 3.5 and 3.6, respectively.

3.4 Remote debugging

The challenge-response protocol described above implements a *remote debugging* process. Remote debugging occurs when the target program runs on a different location, e.g., a remote host. Under our assumptions, BH executes the target program[12] and BI debugs it.

Remote debugging is common and many debug tools support it. However, there is a crucial difference with the (standard, forward) remote debugging process: our debugging procedure proceeds backward. As a matter of fact, the debugging starts from a (buggy) final state and proceeds toward an initial state (*reverse debugging*).

In principle, reverse debugging does not prevent the early disclosure of the execution trace. In fact, in many cases the state of a program is deterministically determined by its predecessors. Hence, BI might infer the predecessor state without interacting with BH.

▶ **Example 2.** Consider again the code of Example 1 and the final state reached when `c = 255`. Such a state is $\sigma = [\mathtt{z} \leftarrow\!\shortmid 0, \mathtt{a} \leftarrow\!\shortmid 256, \mathtt{c} \leftarrow\!\shortmid 255]$. Trivially, since $\sigma(\mathtt{c})$ is defined, the actual parameter of `foo`, i.e., the initial state, is exposed.

To address this issue BH only partially reveals the debug state: she only discloses the variables that are necessary to the current statement, i.e., those occurring in the expressions to be computed.

▶ **Example 3.** We simulate a reverse debug session starting from σ as in Example 2. The state σ refers to the statement `return 1.0/z` (where only the variable `z` appears). Thus, BH sends to BI the state $\sigma_z = [\mathtt{z} \leftarrow\!\shortmid 0]$. In this way, BI effectively verifies that the division by 0 occurs. Still, she cannot easily infer the values of `a` (that determines the value of `z`). As a matter of fact, any state where `a` is larger than 255 is a candidate predecessor. Assuming that each iteration correspond to a single debug step, the next state revealed by BH is $\sigma_a = [\mathtt{a} \leftarrow\!\shortmid 256]$. The debug step succeeds when BI verifies that the execution of the current statement on state σ_a results in state σ_z.

3.5 Challenge generation

As stated above, the challenge is a boolean condition that drives a decision procedure encoded as a smart contract. In particular, given a program state σ, the challenge must precisely

[12] In principle, BH might even reply a recorded execution trace without executing the program. This is also called *Post-mortem* debugging.

characterize a state σ' being a valid predecessor of σ in the debug procedure. Moreover, to solve the challenge, both σ and σ' must belong to the execution trace initially committed by BH (see Section 3.3).

A prominent technique for this task is *backward symbolic execution* [16, 2]. Backward symbolic execution is used to obtain valid preconditions for the execution of a statement starting from its postconditions. This is typically achieved by means of a *weakest precondition* calculus [3]. Briefly, given a program s and a postcondition Q, a weakest precondition is the most general predicate P such that $\models \{P\}s\{Q\}$.

▶ **Example 4.** Consider again the ASCL code of Example 1. The predicate $\texttt{z} \neq \texttt{0}$ is a postcondition for the statement `float z = 255/a`. The weakest precondition for the statement is a predicate P such that $P \Rightarrow \texttt{z} \neq \texttt{0}$. Since $\texttt{z} = \texttt{255/a}$ this becomes $P \Rightarrow \texttt{255/a} \neq \texttt{0}$. Moreover, due to the semantics of the integer division operator in C this is equivalent to $P \Rightarrow \texttt{a} \leq \texttt{255}$. Clearly, the most general (weakest) predicate P that satisfies the implication is $\texttt{a} \leq \texttt{255}$.

To generate a challenge, BI can follow the strategy below. First, BI converts the current debug state to a predicate Q defined as $\bigwedge_{x \in Dom(\sigma)} x = \sigma(x)$. The predicate Q is the precondition to the current debug statement. Also, Q is the postcondition of all the previous statements, i.e., those to be debugged to reach the initial state of the execution. Hence, BI selects a number n of backward steps. From the code, BI extracts all the sequences of statements of length n that can precede the current statement. Via backward symbolic execution on the selected statements, BI computes the weakest preconditions for Q. The resulting predicate is the challenge for BH that she answers by providing the actual state that satisfies the precondition.

▶ **Example 5.** Consider the debug session given in Example 3. The final state σ_z results in the predicate $\texttt{z} = \texttt{0}$. Assuming $n = 1$, the challenge for BH is $\texttt{a} > \texttt{255}$ (trivially from Example 4). Then, BH successfully answers by providing σ_a.

It is evident from the example above that the choice of n is critical. In general, the size of a predicate computed through backward symbolic execution can grow exponentially with n [8]. Intuitively, the exponential blow-up is caused by the conditional statements.

In software verification, large predicates pose serious limitations. Indeed, *satisfiability modulo theories* (SMT) [6] is used to verify whether a certain predicate admits a model, i.e., an assignment of values that satisfy the predicate. The SMT problem is computationally hard, but its complexity varies with the underlying theory. In our context, bit-vectors are the most common theory. The SMT problem for bit-vectors is known to be (in the best case) NP-complete [13]. Nevertheless, this is not a limitation in our context as BH already knows a solution to the challenge, that is the program state that she has committed.

3.6 Smart contracts and blockchain

In this section we describe the structure of the smart contracts used by VeriOSS. There are two smart contracts, i.e., the bounty issuing contract and the partial reward contract. The first one is straightforward. Its role is to describe the bug and the offered reward. The second contract requires more attention. As a matter of fact, it is responsible for the partial rewarding defined in Section 3.3.

Figure 4 shows an example *Solidity* [5] contract for the challenge of Example 5, i.e., $\texttt{a} > \texttt{255}$. The contract handles three pieces of information (lines 2-4), i.e., the address of the bounty hunter, the amount of the reward and an expiration time. The main function of the

```
1  contract PartialReward {
2   address public hunter = /* ... */;
3   uint    public reward = /* ... */;
4   uint    public expire = /* ... */;
5
6   function challenge(bytes4[] state) public {
7    if(decommit(state) && solve(state))
8     hunter.transfer(reward);
9   }
10  function solve(bytes4[] state) private returns (bool) {
11   if(state[0] <= 255) /* a ≤ 255 */ return false;
12   return true;
13  }
14  function decommit(bytes4[] state) private returns (bool)
15  { /* check state hash */ }
16  function timeout() public { require(now >= expire);
17   selfdestruct(this); }
18 }
```

Figure 4 An instance of the partial reward smart contract.

contract is `challenge` (line 6). The hunter invokes the function by providing the program state as a list of bytes. Then, the contract invokes two functions, i.e., `decommit` and `solve` (line 7). The former (line 14) decommits the input state (i.e., it checks its hash code against the list initially provided by the BH). The latter verifies that the provided state is a valid solution to the challenge. If both the checks succeed, the contract transfers the reward to the hunter. The function `solve` (line 10) encodes the challenge. It consists of a sequence of conditional statements. Each statement checks whether a single clause of the challenge is violated. In that case, `solve` returns `false`. When all the checks are passed, the function returns `true`. Finally, the contract has a `timeout` function (line 16) to void it when the deadline expires.

Few aspects of the contract of Figure 4 need a further discussion. In the first place, the structure of function `solve`. Clearly, it is the most expensive function in terms of computation and, since on chain computation is not for free [19], efficiency might be an issue. As highlighted in Section 3.3, checking the solution to a challenge is linear in the number of constraints. However, this number grows exponentially with n. Thus, a proper trade-off must be considered.

4 Incentives

From the economic viewpoint, VeriOSS aims at allowing a profitable trading between a seller (BH) and a buyer (BI) of information (the bug). The protocol of Section 3 can accomplish this goal, but BH and BI may refuse to run it. The main reason is *hold-up* [1], i.e., the buyer can refuse to pay after she leaned the information. This could prevent potentially profitable exchanges due to stall between the seller (who wants to be paid before disclosing the information) and the buyer (who wants to evaluate the information before paying it).

VeriOSS overcomes this issue by delegating the verification of the information and the payment of the reward to a smart contract. By itself, however, this is not sufficient to give BI and BH the correct economic incentives to follow the protocol of Section 3. Below we list the incentives problem faced by BI and BH at every step of the protocol, and how VeriOSS addresses them.

1. Since BI puts forward the reward when publishing the initial bounty contract, the reward offered by BI might be inadequate for BH. However, we expect a round of communication between BI and BH to occur beforehand to ensure that BI and BH agree on the reward. Also, due to the guarantees of the P2K protocol, BI and BH can negotiate under the assumption that the counterpart is honest.
2. The cost of the off-chain computation of BI is not negligible. In particular, computing the weakest preconditions may be computationally hard. For this reason, it is crucial that the information initially disclosed by BH provides a proper incentive to set up the challenges. For instance, BH might need to initially reveal some extra details about the debug trace. What is the right amount of information is an open research question, e.g., see [11].
3. A malicious BI could intentionally craft an incorrect challenge. The main motivation here is inferring as much information as possible from BH's answer. For example, BI might submit an unsatisfiable challenge to make the protocol fail even if the provided answer is correct. In this way, BI may collect the next state without paying the partial reward. However, BH can also compute the weakest preconditions and detect an incorrect challenge. In such a case, she can retire from the protocol with no loss.
4. Even if BH has always answered correctly, BI could decide to interrupt the protocol before the end. For instance, BI may believe that the information still to be released by BH is worth than the remaining reward. This boils down to correctly establishing the partial rewards, so to adequately compensate BH while encouraging BI to continue. As long as they correctly price each iteration, BI is not motivated to interrupt the protocol.[13]
5. BH may attempt to renegotiate the reward after BI computes a challenge, i.e., BH can hold up BI. Indeed, since it is costly, BI may accept to pay an higher partial reward to avoid recomputing the challenge. Note, however, that the total reward is established at the beginning of the protocol. Hence, an honest BH would no obtain an higher total payment. Since it reveals that BH is malicious, no BH (malicious or honest) is motivated to renegotiate.

Finally, note that the above analysis assumes the presence of a single BH and a single BI. This is not the case in general. The presence of other BIs and BHs may affect the incentives faced by the protocol's participants, and hence the performance of the protocol. We discuss this issue in the next section.

5 Discussion

In this section we provide a detailed discussion on some aspects that may affect the implementation of VeriOSS, some open issues and future developments.

5.1 Implementation details

The implementation will need to address issues about some aspects we left abstract in the previous sections. A first issue concerns how to represent the commitment trace of BH. Intuitively, this trace can be obtained by computing the hashes of each state in the original debug trace. However, this may be impractical because of the length of the debug trace. Thus, we need an implementation that compresses these traces without compromising the validity of the protocol.

[13] This issue can also be more directly addressed by using an escrow (see Section 5.2).

Another issue is about the number n of iterations of the challenge-response protocol. This choice is quite critical: different choices of n may lead to different cost in term of (i) cryptocurrency paid by the parties and (ii) efficiency of the protocol. Finding a good trade-off is left as future work.

5.2 Threat model

The design of VeriOSS is based on a threat model where both BI and BH do not trust each other and both may be malicious. On the one hand, a malicious BI aims at collecting information about a bug without paying the corresponding reward. Our protocol opposes this behavior and forces BI to behave honestly by (i) requiring a precise specification of the eligible bugs (Section 3), (ii) increasing the bargaining power of the BH, (iii) providing the partial reward mechanism in which a small portions of the bounty is paid in each iteration for each piece of revealed information (Section 3.3).

On the other hand, a malicious BH aims at obtaining an undue reward. For instance, BH might submit a partial or a false bug trace during the remote debug protocol. Also, a malicious BH could attempt a *reply attack* by re-submitting an old, already paid trace. VeriOSS protects honest BIs against malicious BHs by (i) establishing a commitment phase (Section 3.3), (ii) providing a challenges-response protocol. In particular, the trace commitment ensures that past traces are automatically detected, e.g., because they terminate with the very same state of a previously executed trace. Instead, the challenge-response protocol ensures that each step of the trace is correct.

In addition, the protocol can be easily extended by introducing a second smart contract acting as an escrow that collects all the partial rewards and then forwards them to the BH only if the bug is entirely disclosed. In this way, a malicious BH cannot obtain any partial reward and, at the same time, a malicious BI cannot gain by strategically interrupting the protocol. As future work, we plan to further study the robustness of our mechanisms against this attacker model.

5.3 Future extension

Here, we outline some future directions for the development of VeriOSS.

The current design of VeriOSS allows a single BH and a single BI to efficiently exchange the bug trace against a reward. However, this is just an intermediate goal, because the bug bounty market consists of several actors. Currently, our bug disclosure process in exclusive between one BI and one BH. Instead, "open" sessions might allow other parties to interact, e.g., by offering a better reward.

The blockchain used by VeriOSS allows parties that do not know or trust each other to interact. Sometimes this is not desirable, e.g., when knowing the identity of the BI necessary to discriminate between legitimate companies and malicious actors. As a mitigating, we could allow the BI creating the smart contract to "sign" it using its private keys, therefore allowing everybody to verify that a given challenge was indeed created by a reputable BI. This, of course, would not prevent malicious actors from creating their own smart contracts using VeriOSS, but it would make public that a given (supposedly malicious) actor posted a challenge and obtained information regarding a bug. Discriminating between legitimate and malicious BIs is a future work.

Also, many different firms may benefit from discovering and fixing bugs in FOSS. This gives rise to what is known as "free rider" problem. The maximum payment a BH can receive depends on the willingness to pay for the bug report of the firm valuing it the most. Such

a payment can be significantly lower than the overall benefit of finding the bug. VeriOSS can include a mechanism to aggregate rewards from several BIs. For instance, this can be achieved by introducing *reward rise contracts* that BIs can use to offer a further incentive toward the disclosure of a certain bug. Again, this is future work.

Finally, the presence of other actors is also relevant for the issue of "responsible disclosure". In most bug bounty programs, all parties are contractually forbidden from publicly disclosing the bug for a period of time. Such a time span may be legally imposed and it is intended to give the BI enough time to fix the bug. In VeriOSS, instead, the bug is immediately public, which implies that a malicious actor could exploit it before the BI manages to implement a remediation. This is also a direction where we plan to improve the protocol.

5.4 Limitations

Here, we briefly discuss some limitations of our proposal. In the design of VeriOSS we assume that BH has a copy of the software to test. This is not a problem for mobile apps, or desktop software that the BH can download from the network and run on her machines. However, when the target software is a web application or web service our remote debugging protocol cannot be applied as is. Indeed, in those situations the BH can mainly interact with the software by providing inputs and receiving outputs (a.k.a. black box testing). Hence, BH does not have access to the full program state, which is partially stored on a remote server.

Another limitation is the assumption that a bounty is only issued for a bug, i.e., a faulty state of the target program. Often, a BI only offers a reward for a bug that actually impacts on the security of the software. Said differently, a BI might ask for an exploit exposing her software to concrete attacks, e.g., data breaches. Thus, the offered reward depends on the value of the assets that an attacker can steal or compromise. At the moment VeriOSS does not support this kind of bounty programs. Furthermore, although formal specification languages can precisely characterize a failure condition, one could argue that some types of bugs cannot be expressed (easily or even at all). For instance, think about the remote code execution caused by a ROP chain [17]. For these reasons, we plan to introduce multiple languages for the specification of bugs and exploits. The main requirement for the bug definition languages is that they must provide a sound eligibility check (so that BI cannot repudiate an eligible bug) and support the challenge-response process.

6 Related work

VeriOSS is made of different components each based on a specific technology. Here, we follow the line of Section 3 and compare each component of VeriOSS with similar proposals.

6.1 Remote attestation

Remote attestation [4] allows a remote host (*the challenger*) to authenticate the hardware and software configuration of another remote host (*the attestator*) which is charge of performing some computation. The attestator is equipped with a suitable *Trusted Platform Module* (TPM) chip, which she uses to attest the states of its software components to the challenger. Typically, this verification is based on digital signatures, i.e., the challenger only verifies that the signatures sent by the attestator are as expected. This basic attestation mechanism can be used as a building block to check other security properties. For example, [9] proposes the implementation of a trusted virtual machine that not only allows running a program but also attesting to a remote entity that the running program satisfies a given set of security properties at run time.

At a first sight, one may think that the challenge-response interaction protocol of Section 3.3 may be implemented using remote attestation. However, this is not the case because mainly remote attestation only allows BI to avoid a malicious BH, but not vice versa. Furthermore, remote attestation requires that BH is equipped with a specific hardware, i.e., TPM chip, that increase the cost of entering the market. Our protocol, instead, provides a mechanism to protect both participants and does not require any specific hardware.

6.2 Remote debugging

Modern development environment allows debugging applications remotely. This is very useful when the development system is different than the production one. The underlying idea is that the debugger is installed on the production server and that it provides a network channel for interacting with the debbuged program. The programmer uses a client that completely abstract the interactions through the network. In this way, debugging a program remotely is almost the same as doing it locally. In particular, this means that the client can stepwise run the program and can inspect its memory.

There are at least two crucial differences between a standard remote debugging and the approach described in Section 3.4. The first is that in standard remote debugging there is only an agent interacting with the program that is the client (the server only makes available the state of the program to the client); then, the client and the server trust each other, or both are under the same administrative domain. Whereas in our setting, BI and BH are two different agents in the system that does not trust each other.

The second important difference is that the standard remote debugging proceeds forwards and the client can access the entire state of the execution. In our approach, instead, the debugging proceeds backward and the BI can access only a specific part of the state of the execution.

6.3 Information flow

Information flow control (IFC) is a mandatory access control mechanism that enforces some restrictions on a piece of data and on all data derived from it. It was introduced in [7] as mechanism to enforce *non-interference* across security levels. IFC is continuously enforced at every information exchange. The underlying idea is that each piece of information is associated with a policy (tags working as metadata) describing its level of secrecy and integrity. Moreover, also entities of a system are associated with a security level, describing the sensitivity of the data they are allowed to handle. The mechanism guarantees that entities with a lower security clearance cannot read/write up to information with higher security level. Symmetrically, it also ensures that entities with higher security clearance cannot write down by making a disclosure of information.

During our remote debugging process the BH should not reveal too much information about the execution state, so that a malicious BI cannot reconstruct all the execution state. To do that, our protocol prescribes that BH shares only a part of the state. To determine which part of the state to be disclosed, the BH should follows an approach based on IFC.

6.4 Secure multi-party computation

A multi-party computation occurs when two or more entities join together to compute a certain function f. More precisely, consider n parties $P_1, \ldots, P_n$, each with its own input x_i, which want to compute $f(x_1, \ldots, x_n)$. A *secure multi-party computation* [10] is a multi-party computation where each participant P_i aims to preserve the privacy of its input x_i.

Our challenge-response protocol fits this setting where the function f to compute consists in the challenge verification process. Indeed, the BI provides as input a pair q made of the challenge and of the commitment contract; whereas the BH provides the corresponding state σ. The function f then perform the relevant checks and return a pair q' containing the reward for BH and the computation state for the BI. However, differently from the case of the secure multi-party computation, the input of BH is private, whereas the one of BI is public (and indeed published on a blockchain).

6.5 Information sharing

The inefficiencies of bug bounty programs are commont to all markets for information and have been known at least since [1]. Several authors have studied mechanisms to resolve these inefficiencies. The most closely related work is [11], which proposes a protocol in which the seller of information sustains several tests. Every time a test is successfully completed, the buyer sends the seller a partial payment. In their baseline model, the tests are such that if the seller really has the piece of information, then she passes the test. If she does not, then she can complete the test with probability $p < 1$, where lower values of p correspond to more stringent (and hence more informative) tests. In the first round of the protocol, the seller reveals some information by sustaining a test for free. After observing the result of the test, the buyer updates his belief regarding whether the seller has the piece of information, and with it the expected benefit of learning it. The buyer then sends a payment to the seller who sustains another test, and so on. The information is thus revealed in stage (by sustaining each test) and to each revelation stage corresponds a partial payment.

Crucially, [11] assumes a total lack of commitment: the buyer is free to withhold the payment to the seller, even after the seller passes the test. In the equilibrium with information revelation the prospect of learning additional information motivates the buyer to follow the protocol. But many other equilibria exist, including some in which no information is ever revealed.

The part of VeriOSS that is most closely related to [11] is the initial exchange of information, in which the BH reveals the initial state. The reason is that, at this stage, the BI is under no obligation to set up the smart contract and start the protocol. This lack of commitment implies that the intuition in [11] applies here as well. However, once the smart contract is set up and the iteration of challenge/response begins, [11] ceases to be relevant. The reason is that the BI can commit to pay the BH if and only if the BH has the correct piece of information. The fact that information is revealed in stages (with corresponding partial payments) is done exclusively for practical reasons. As already discussed, the protocol could run with a single test and a single answer revealing the entire trace, but crafting such test is computationally very expensive. For this reason the information generation protocol is split in different stages.

7 Conclusion

In this paper we presented VeriOSS, a novel paradigm for the construction of bug bounty programs. VeriOSS-based programs provide concrete guarantees that a bounty hunter will receive her rewards without trusting the bounty issuer. Together with other relevant properties natively supported by the blockchain, we expect this to favor the flourishing of the bug bounty market.

References

1 Kenneth Joseph Arrow. Economic welfare and the allocation of resources for invention. In *Readings in industrial economics*, pages 219–236. Springer, 1972.

2 Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

3 Marcello M. Bonsangue and Joost N. Kok. The weakest precondition calculus: Recursion and duality. *Form. Asp. Comput.*, 6(1):788–800, November 1994.

4 George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian O'Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of remote attestation. *Int. J. Inf. Sec.*, 10(2):63–81, 2011.

5 Chris Dannen. *Introducing Ethereum and Solidity*. Apress, Berkely, CA, USA, 1st edition, 2017.

6 Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

7 Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

8 Cormac Flanagan, Cormac Flanagan, and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 193–205. ACM, 2001.

9 Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3*, VM'04, pages 3–3. USENIX Association, 2004.

10 Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

11 Johannes Hörner and Andrzej Skrzypacz. Selling information. *Journal of Political Economy*, 124(6):1515–1562, 2016.

12 Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015.

13 Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In Pascal Fontaine and Amit Goel, editors, *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories*, volume 20 of *EPiC Series in Computing*, pages 44–56. EasyChair, 2013.

14 Bertrand Meyer. Contract-driven development. In *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, pages 11–11, Berlin, Heidelberg, 2007. Springer-Verlag.

15 Aybek Mukhamedov, Steve Kremer, and Eike Ritter. Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model. In Andrew S. Patrick and Moti Yung, editors, *Financial Cryptography and Data Security*, pages 255–269, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

16 Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515. ACM, 2011.

17 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

18 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

19 Daniel Davis Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, 2014. (White paper).