

## VeriOSS: Using the Blockchain to Foster Bug Bounty Programs

Questa è la versione preprint della seguente opera:

*Original*

VeriOSS: Using the Blockchain to Foster Bug Bounty Programs / Canidio, Andrea; Costa, Gabriele; Galletta, Letterio. - 82:6(2021), pp. 1-14. ( 2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020)).

*Availability:*

This version is available at: 20.500.11771/15589

*Publisher:*

*Published*

DOI:

*Terms of use:*

This publication is made accessible in accordance with the terms for deposit in the institutional repository, as defined by the IMT School for Advanced Studies Lucca's Open Access Policy. ([https://library.imtlucca.it/sites/default/files/regolamento-policy-open-access-imtlib\\_0.pdf](https://library.imtlucca.it/sites/default/files/regolamento-policy-open-access-imtlib_0.pdf)).

Si prega di consultare le pagine informative dell'editore relative alle politiche di autoarchiviazione.

(Article begins on next page)

# VeriOSS: using the Blockchain to Foster Bug Bounty Programs

Position paper

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.CVIT.2016.23

**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.<sup>1</sup> 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.<sup>2</sup>

---

<sup>1</sup> <https://www.forbes.com/sites/taylorarmerding/2019/01/09/the-future-of-open-source-software-more-of-everything/>

<sup>2</sup> <https://www.google.com/about/appsecurity/chrome-rewards>

## 23:2 VeriOSS: using the Blockchain to Foster Bug Bounty Programs

40 Often, BI is the software developer or owner, e.g., Google in the example above. However,  
41 a bounty can be also issued for third-party software. This is the case for FOSS components  
42 involved in some critical systems, either open or proprietary. A prominent example of  
43 bounties for third-parties software is the *Free and Open Source Software Audit* (FOSSA)  
44 project, sponsored by the European Commission and offering bounties of up to several  
45 hundreds of thousands of euros for vulnerabilities discovered in 14 major FOSS.<sup>3</sup> According  
46 to the project executives, FOSSA is a response to *Heartbleed*, a severe security vulnerability  
47 that affected OpenSSL in 2014.<sup>4</sup>

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

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

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

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

---

<sup>3</sup> <https://juliareda.eu/fossa>

<sup>4</sup> <http://heartbleed.com>

<sup>5</sup> <https://sites.google.com/site/bughunteruniversity/behind-the-scenes/charts/2016>

<sup>6</sup> 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/>.

<sup>7</sup> <https://www.hackerone.com>

<sup>8</sup> <https://www.intigriti.com>

<sup>9</sup> 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  $c_1, \dots, c_k$ . 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  $\sigma$  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  $\langle \sigma, s \rangle \rightarrow \langle \sigma', s' \rangle$  to denote that in the state  $\sigma$  the program  $s$  executes one of its statements, becomes  $s'$  and modifies the state in  $\sigma'$ . For brevity, we write  $\langle \sigma, s \rangle \rightarrow^* \langle \sigma', s' \rangle$  as a shorthand for a finite sequence of steps  $\langle \sigma, s \rangle \rightarrow \dots \rightarrow \langle \sigma', s' \rangle$ . Moreover, when a computation terminates, i.e., the destination configuration contains an empty sequence of statements, we simply write the final state as  $\langle \sigma, s \rangle \rightarrow^* \sigma'$ . 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  $\forall \sigma, \sigma'. P(\sigma) \wedge \langle \sigma, s \rangle \rightarrow^* \sigma' \Rightarrow Q(\sigma')$ . 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.

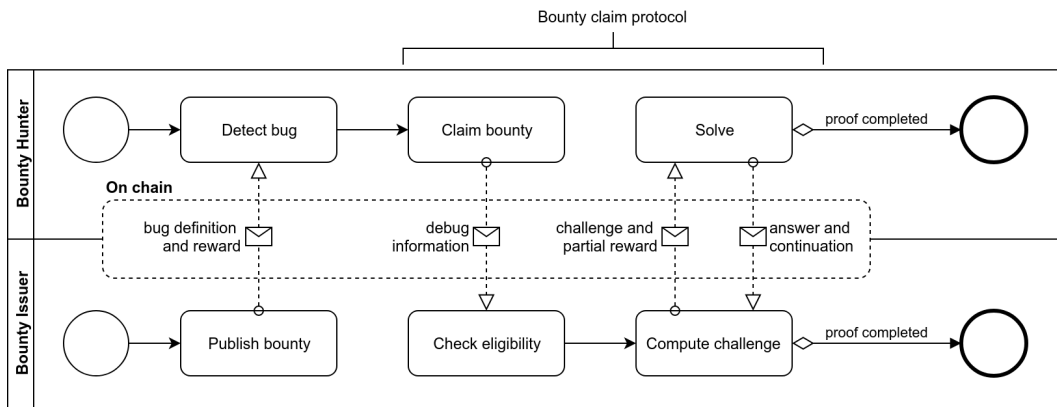
## 3 VeriOSS

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.

### 121 3.2 Bug specification

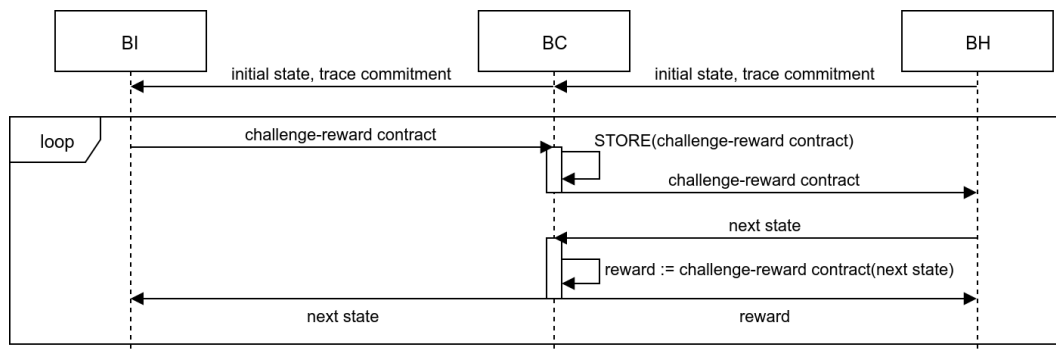
122 When publishing a bounty, BI has to provide a rigorous description of the bugs that are  
 123 eligible for the reward. Such a description contractualizes the commitment of BI to pay for  
 124 a compatible bug. Some classification techniques exist to define bugs and vulnerabilities.  
 125 For instance,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

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

141 ► **Example 1.** Consider the C code of Figure 2. If we are interested in spotting out divisions  
 142 by 0, there are two candidate instructions, i.e., the assignment to *z* and the `return` statement.

<sup>10</sup><https://www.first.org/cvss/specification-document>

<sup>11</sup>[https://cwe.mitre.org/cwss/cwss\\_v1.0.1.html](https://cwe.mitre.org/cwss/cwss_v1.0.1.html)



■ **Figure 3** The P2K protocol message sequence diagram.

143 In terms of properties to be satisfied, the preconditions for the two statements are  $a \neq 0$  and  
 144  $z \neq 0$ , respectively. In ASCL the corresponding assertions are placed right before the target  
 145 instructions as in Figure 2. Here, the bug is exposed (only) when  $c = 255$ . As a matter of  
 146 fact, due to the integer division  $255/256$ , 0 is assigned to  $z$ , so violating the second assertion.

### 147 3.3 Challenge-response interaction

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

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

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

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

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

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

### 189 3.4 Remote debugging

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

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

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

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

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

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

### 214 3.5 Challenge generation

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

---

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

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

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

225 ► **Example 4.** Consider again the ASCL code of Example 1. The predicate  $z \neq 0$  is  
 226 a postcondition for the statement `float z = 255/a`. The weakest precondition for the  
 227 statement is a predicate  $P$  such that  $P \Rightarrow z \neq 0$ . Since  $z = 255/a$  this becomes  $P \Rightarrow$   
 228  $255/a \neq 0$ . Moreover, due to the semantics of the integer division operator in C this is  
 229 equivalent to  $P \Rightarrow a \leq 255$ . Clearly, the most general (weakest) predicate  $P$  that satisfies  
 230 the implication is  $a \leq 255$ .

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

240 ► **Example 5.** Consider the debug session given in Example 3. The final state  $\sigma_z$  results  
 241 in the predicate  $z = 0$ . Assuming  $n = 1$ , the challenge for BH is  $a > 255$  (trivially from  
 242 Example 4). Then, BH successfully answers by providing  $\sigma_a$ .

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

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

### 253 3.6 Smart contracts and blockchain

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

259 Figure 4 shows an example *Solidity* [5] contract for the challenge of Example 5, i.e.,  
 260  $a > 255$ . The contract handles three pieces of information (lines 2-4), i.e., the address of the  
 261 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.

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

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

## 278 4 Incentives

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

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

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

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

## 322 **5 Discussion**

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

### 325 **5.1 Implementation details**

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

---

<sup>13</sup>This issue can also be more directly addressed by using an escrow (see Section 5.2).

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

## 336 5.2 Threat model

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

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

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

## 358 5.3 Future extension

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

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

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

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

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

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

## 387 5.4 Limitations

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

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

## 407 6 Related work

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

### 410 6.1 Remote attestation

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

419 also attesting to a remote entity that the running program satisfies a given set of security  
420 properties at run time.

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

## 427 6.2 Remote debugging

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

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

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

## 445 6.3 Information flow

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

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

## 460 6.4 Secure multi-party computation

461 A multi-party computation occurs when two or more entities join together to compute a  
462 certain function  $f$ . More precisely, consider  $n$  parties  $P_1, \dots, P_n$ , each with its own input  $x_i$ ,

463 which want to compute  $f(x_1, \dots, x_n)$ . A *secure multi-party computation* [10] is a multi-party  
464 computation where each participant  $P_i$  aims to preserve the privacy of its input  $x_i$ .

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

## 472 6.5 Information sharing

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

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

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

## 502 7 Conclusion

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

## 508 — References

- 509 1 Kenneth Joseph Arrow. Economic welfare and the allocation of resources for invention. In  
510 *Readings in industrial economics*, pages 219–236. Springer, 1972.
- 511 2 Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi.  
512 A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.
- 513 3 Marcello M. Bonsangue and Joost N. Kok. The weakest precondition calculus: Recursion and  
514 duality. *Form. Asp. Comput.*, 6(1):788–800, November 1994.
- 515 4 George Coker, Joshua D. Guttman, Peter Loscocco, Amy L. Herzog, Jonathan K. Millen, Brian  
516 O’Hanlon, John D. Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of  
517 remote attestation. *Int. J. Inf. Sec.*, 10(2):63–81, 2011.
- 518 5 Chris Dannen. *Introducing Ethereum and Solidity*. Apress, Berkely, CA, USA, 1st edition,  
519 2017.
- 520 6 Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: Introduction and  
521 applications. *Commun. ACM*, 54(9):69–77, September 2011.
- 522 7 Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243,  
523 May 1976.
- 524 8 Cormac Flanagan, Cormac Flanagan, and James B. Saxe. Avoiding exponential explosion:  
525 Generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-*  
526 *SIGACT Symposium on Principles of Programming Languages*, POPL ’01, pages 193–205.  
527 ACM, 2001.
- 528 9 Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual  
529 machine directed approach to trusted computing. In *Proceedings of the 3rd Conference on*  
530 *Virtual Machine Research And Technology Symposium - Volume 3*, VM’04, pages 3–3. USENIX  
531 Association, 2004.
- 532 10 Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and*  
533 *Constructions*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.
- 534 11 Johannes Hörner and Andrzej Skrzypacz. Selling information. *Journal of Political Economy*,  
535 124(6):1515–1562, 2016.
- 536 12 Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski.  
537 Framac: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May  
538 2015.
- 539 13 Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size  
540 bit-vector logics with binary encoded bit-width. In Pascal Fontaine and Amit Goel, editors,  
541 *SMT 2012. 10th International Workshop on Satisfiability Modulo Theories*, volume 20 of *EPiC*  
542 *Series in Computing*, pages 44–56. EasyChair, 2013.
- 543 14 Bertrand Meyer. Contract-driven development. In *Proceedings of the 10th International*  
544 *Conference on Fundamental Approaches to Software Engineering*, FASE’07, pages 11–11,  
545 Berlin, Heidelberg, 2007. Springer-Verlag.
- 546 15 Aybek Mukhamedov, Steve Kremer, and Eike Ritter. Analysis of a multi-party fair exchange  
547 protocol and formal proof of correctness in the strand space model. In Andrew S. Patrick  
548 and Moti Yung, editors, *Financial Cryptography and Data Security*, pages 255–269, Berlin,  
549 Heidelberg, 2005. Springer Berlin Heidelberg.
- 550 16 Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental  
551 symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming*  
552 *Language Design and Implementation*, PLDI ’11, pages 504–515. ACM, 2011.
- 553 17 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented program-  
554 ming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34,  
555 March 2012.
- 556 18 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT  
557 Press, Cambridge, MA, USA, 1993.
- 558 19 Daniel Davis Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. 2014.  
559 (White paper).